University of Alberta

Sandwich: A Personal Web Assistant Framework

by

Wendy Liew ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree Master of Science.

Department of Computing Science

Edmonton, Alberta

Fall 1999

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisitions et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-47057-1

Canada

University of Alberta

Library Release Form

Name of Author: Wendy Liew

Title of Thesis: *Sandwich*: A Personal Web Assistant Framework

Degree: Master of Science

Year this Degree Granted: 1999

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly, or scientific research purposes only.

The Author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any form whatever without the author's prior written permission.

Wendy Liew
9215-172 Ave
Edmonton, AB
T5Z 2M8

Date: Sep 16, 1999

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled ***Sandwich*: A Personal Web Assistant Framework** by Wendy Liew in partial fulfillment of the requirements for the degree of Master of Science.

Dr. Paul Sorenson (Co-supervisor)

Dr. Jim Hoover (Co-supervisor)

Dr. Eleni Stroulia (Examiner)

Dr. Michael Barrett (External)

Date: Sep 8, 1999

To my family,

they make my achievements worthwhile

Abstract

The plethora of information available on the Internet, coupled with its high degree of informal interconnectedness, makes it difficult to perform advanced web browsing. Examples are the relatively poor support for managing browsing history and the delegating of simple, reactive web-processing tasks. This thesis explores ways to mitigate this problem through the use of a Java-based personal web proxy server. This proxy server is granted the privilege to snoop into and keep track of a user's daily web activities. Assuming this basic capability, an application framework (***Sandwich***) is developed to operate as a rudimentary web proxy server that can be enhanced to support advanced web activities for a user.

An application framework embodies a generic design for a family of applications in a given domain. At present, there is much academic and industry interest in application frameworks because of their high potential for software reuse. ***Sandwich***, the application framework developed in support of personal web assistants, provides basic functions and hooks for plugging in assistants in support of advanced web browsing. In particular, based on the thesis research, two categories of assistants are identified and supported: observing and delegating assistants. An observing assistant monitors, measures, and reports on the user's browsing activities. An example is a advanced history assistant that tracks all sites a user has visited and the time these sites were last visited, and then produces a statistical report of this history. A delegating assistant monitors and acts on the user's behalf. An example is an auto posting assistant that monitors regular stock quote requests and acts on the user's behalf upon next request.

The major contributions of this work are

- the identification of the key (or basic) functionality that a personal web assistant should have,
- the encapsulation of this functionality in an extendible framework known as ***Sandwich***,
- the demonstration of the power of this approach by implementing several personal assistants based on the hooks for extending the ***Sandwich*** framework, and
- the provision of hook documentation for the framework in order to enhance the usability and evolution of ***Sandwich*** and to assist in exercising and further validating the hooks model as developed by G. Froehlich in his Ph.D. thesis

Acknowledgments

The list of people I want to thank and acknowledge is a long one, and I would like to apologize to those I have missed. First and foremost, I wish to thank Professors Jim Hoover and Paul Sorenson as my supervisors. They inspired my interest and guided me through this dissertation, even during their sabbatical period. I would also like to thank Eleni Stroulia, Michael Barrett, and Ehab Elmallah for having served on my committee.

Next special thanks go to Garry Froehlich, whose Ph.D. thesis gave me a head start. I am grateful to all other members of the Software Engineering Research Lab (SERL) for their interest in my work, for their useful suggestions and for making the lab such a great place. In particular, thank you to Professor Eleni Stroulia for her advice on the Artificial Intelligent area that I touched on. Also, thank you to Tony Oleksky for taking time off his busy schedule to attend some of my initial brainstorming meetings.

Thank you again to Professor Jim Hoover and Paul Sorenson for giving me the flexibility to pursue my MSc studies while working part-time. I am also very thankful to my managers and colleagues at work. During the course of my MSc studies, I was working as a Java Consultant on two projects. I would like to thank from the Reflective Systems Group Inc. (RSG) John deHaan, who supervised me in the TELUS Multimedia project, and Terry Butt, who supervised me in the IBM Global Services project. I especially appreciate RSG for the flexible work schedule. Both of these projects have coincidentally strengthened my technical skills, which proved very helpful during my research.

Lastly, I would also like to thank my boyfriend, Ernest Siu, and all of my family members for their constant support, endurance, and patience.

Table of Contents

1. INTRODUCTION **1**

1.1. MOTIVATION 1

1.2. THESIS ROADMAP 3

2. BACKGROUND, RELATED WORK AND THESIS APPROACH **4**

2.1. SYSTEM WIDE ASSISTANTS 4

2.2. PERSONAL ASSISTANTS 7

2.2.1 Client Side *7*

2.2.2 Server Side *7*

2.2.3 In-Between *7*

2.2.4 Approach Summary *8*

2.3. AGENT SYSTEMS 9

2.3.1 Agent Definition *9*

2.3.2 Agent Categories *10*

2.3.3 Agent System Projects *11*

2.3.4 Assistant *13*

2.4. HTTP 13

2.4.1 Example *14*

2.4.2 Approach Summary *15*

3. HOOKING INTO APPLICATION FRAMEWORKS **16**

3.1. OBJECT TECHNOLOGY AND REUSE 16

3.1.1 Types of Frameworks and Examples *17*

3.1.2 Current State of Reuse Practice *17*

3.1.3 Approach Summary *18*

3.2. APPLICATION FRAMEWORKS 18

3.2.1 Application Frameworks versus Applications *19*

3.2.2 The Pros and Cons of Application Frameworks *20*

3.2.3 Framework Documentation *21*

3.3. THE HOOK MODEL 22

4. *SANDWICH* **25**

4.1. OVERVIEW 25

4.2. *SANDWICH* ACTORS 27

4.2.1 Framework Designers/Developers *27*
4.2.2 Framework Users (Assistant Developers) *27*
4.2.3 Framework Maintainers *27*
4.2.4 End Users *27*
4.3. USE CASES 28
4.3.1 Origin and the Current State-of-the-Art *28*
4.3.2 Introduction *28*
4.3.3 Applying Use Cases *29*
4.3.4 Application Family Use Cases *30*
4.3.5 Sandwich Use Cases *43*
4.4. SANDWICH SUBSYSTEM, DESIGN, HOOKS AND RATIONALE 51
4.4.1 Sandwich *Startup* *54*
4.4.2 HTTP Proxy *57*
4.4.3 Observing Assistant *63*
4.4.4 Delegating Assistant *75*
4.4.5 Administrative Application (Sandwich Interface) *94*
4.4.6 Persistence *102*
4.4.7 Logging *108*
4.4.8 Regular Expression *110*
4.4.9 HTTP Support *112*
4.5. STEPS IN CREATING NEW SANDWICH ASSISTANT 114
4.6. OTHER CONSIDERATIONS 115
4.6.1 Assistant that is Observing and Delegating *115*
4.6.2 Request and Response Delegates Pair *116*
5. EVALUATION **117**
5.1. THE HOOK MODEL 117
5.2. APPLYING UML 122
5.3. SANDWICH VERSUS WEBBY 124
6. CONCLUSIONS AND FUTURE WORK **134**
7. REFERENCES **137**
APPENDIX A : HTTP **141**
APPENDIX B : META TAGS **147**
APPENDIX C : TOOLS AND STANDARDS DEPLOYED **148**

APPENDIX D : UML NOTATION 149

APPENDIX E : SANDWICH PROPERTIES FILES 152

List of Tables

Table 1: Proxy Categories and Examples 6

Table 2: Agent Related Projects 12

Table 3: Hook Template 22

Table 4: *Sandwich* and Webby 127

List of Figures

Figure 1: Before and After View of an End User Browsing the WWW with a Pool of Personal Assistants.....2

Figure 2: A Primitive Web Infrastructure.....5

Figure 3: Web Infrastructure That Deploys System Wide Proxy.....5

Figure 4: *Sandwich* Infrastructure.....9

Figure 5: HTTP Request and Response.....13

Figure 6: Reuse Forms and Levels.....16

Figure 7: Hooks and Hot Spots.....19

Figure 8: Groups and Types of Assistants.....26

Figure 9: Example of a Use Case Diagram.....28

Figure 10: *Sandwich* Use Cases.....43

Figure 11: *Sandwich* Subsystem Architecture.....51

Figure 12: *Sandwich* Interface, the Administrative Application.....52

Figure 13: UML Stereotypes Deployed.....54

Figure 14: *Sandwich* Startup Dynamic View.....55

Figure 15: *Sandwich* Startup Static View.....56

Figure 16: HttpProxy Startup Dynamic View.....59

Figure 17: HttpProxy Startup Static View.....60

Figure 18: HttpProxy Servicing Request Dynamic View.....62

Figure 19: HttpProxy Servicing Request Static View.....63

Figure 20: Data for Observing Assistants.....64

Figure 21: Observing Assistant Initialization.....66

Figure 22: Observing Assistant Notification Dynamic View.....68

Figure 23: Observing Assistants Static View.....71

Figure 24: Request Delegating Assistant.....76

Figure 25: Response Delegating Assistant.....76

Figure 26: Request Delegate Initialization.....77

Figure 27: Response Delegate Initialization.....78

Figure 28: Default DirectHttpDelegate (High Level).....79

Figure 29: Default *DirectHttpDelegate* (Object Level).....81

Figure 30: Multiple Delegating Assistants.....82

Figure 31: Delegate Priority Queues.....83

Figure 32: Multiple Request Delegates 84
Figure 33: Multiple Response Delegates 86
Figure 34: Delegating Assistant Static View 87
Figure 35: Administrative Application and HttpProxy 95
Figure 36: Administrative Application and Assistants 97
Figure 37: Administrative Component Static View 99
Figure 38: Enhanced History Assistant Result Window 102
Figure 39: Persistence 103
Figure 40: Logging Support 108
Figure 41: Regular Expression 111
Figure 42: HTTP Support 112
Figure 43: Assistant that is both Observing and Delegating 115
Figure 44: Request and Response Delegates Pair 116
Figure 45: Hook Dependencies and Optional Paths 120

List of Hooks

Hook 1: New Observing Assistant Hook 73
Hook 2: New Request Delegating Assistant Hook 89
Hook 3: New Response Delegating Assistant Hook 93
Hook 4: New Administrative Application Hook 100
Hook 5: New Assistant Result Window Hook 101
Hook 6: Reading Property File Hook 106
Hook 7: New Line File Data Hook 107
Hook 8: Reading Line File Hook 108
Hook 9: New Logger Hook 110
Hook 10: New Regular Expression Adapter Hook 112
Hook 11: New HTTP Header Hook 113

1. Introduction

1.1. Motivation

In the past five years, the Internet has gained tremendous popularity, and the volume of information available on the web is now becoming overwhelming for an average person to make optimal and effective use of. As the web continues to grow, there is a need to add advanced and specialized capabilities to web browsing. Personal web assistants can help to track and manage this vast store of information. The following describes two examples where personal web assistantship can be useful.

Example 1:

On day one, Alice, an assistant epidemiologist, browses the web for health-related news. She encounters a few sites that provide statistical information on the disease Hepatitis C and reads them through briefly. On day two, coincidentally, Alice's boss asks her to give an updated report on Hepatitis C. She recalls that she has read some useful information on day one, but, unfortunately, she has forgotten to bookmark the sites. Because of insufficient information tracking, Alice is unable to recall all the information that she read and the sites she visited.

If she had had an enhanced history assistant working with her on day one, she would have been fine. This assistant *observes* all pages she has browsed, remembers the URLs and allows Alice navigate to a visited URL.

Example 2:

Another commonly encountered frustration of web users is the process of filling out a form. For example, Michael visits the Check Free Quote Server page at least once a day to get the final quotes on the twenty stocks that he is interested in. Everyday, he routinely does the following:

1. Uses his bookmark to reach the Check Free Quote Server site.
2. Enters five stock symbols and then presses the "submit" button.
3. Gets the response back, e.g. closing prices and traded volume of the stocks.

Check Free restricts each request to five stock symbols; thus, to obtain the twenty stock quotes Michael has to repeat the above steps 2 and 3 four times.

Having an auto post assistant can reduce this aggravation. Specifically, the assistant reduces the above steps to a "select and go" delegation step. To elaborate, Michael will first be prompted by an auto post assistant with a list of four entries, with each entry containing five stock quotes. He confirms the delegation process by selecting one of the entries and then clicking the "confirm" button.

The following figure illustrates a before and after view when personal web assistantship is in place to control web browsing activities.

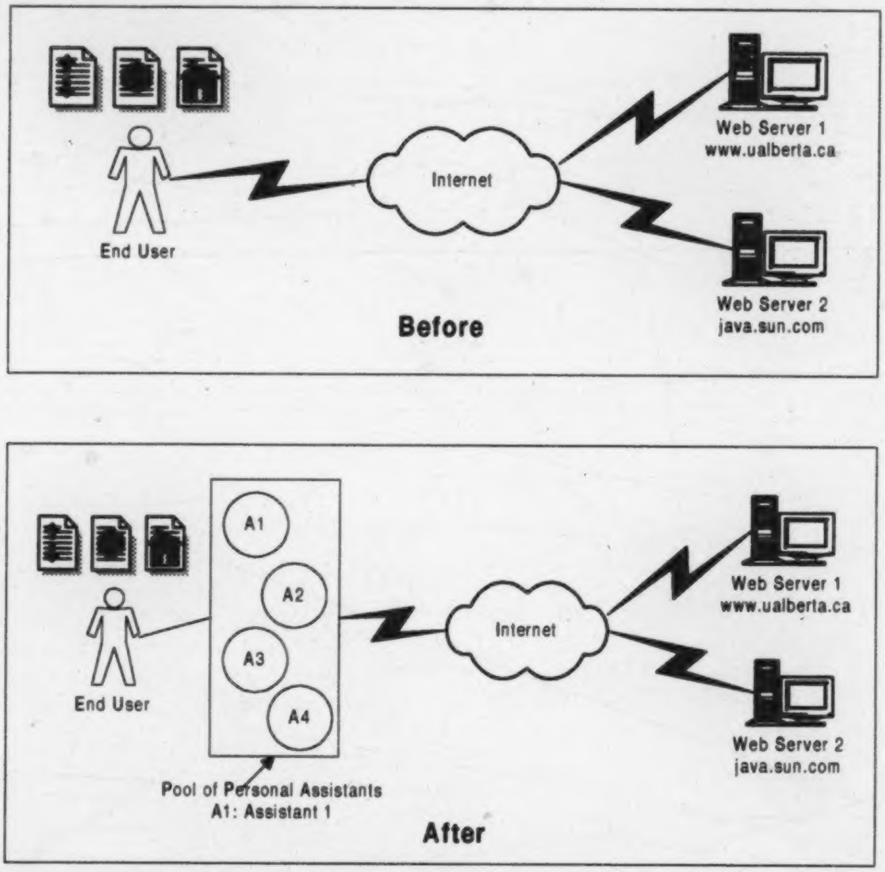

Web Server 1
www.ualberta.ca
Internet
End User
Web Server 2
java.sun.com
Before
A1
A2
A3
A4
Web Server 1
www.ualberta.ca
Internet
End User
Web Server 2
java.sun.com
Pool of Personal Assistants
A1: Assistant 1
After

Figure 1: Before and After View of an End User Browsing the WWW with a Pool of Personal Assistants

This thesis explores the possibility of adding personal web assistantship capabilities embedded into a rudimentary personal proxy server that has been granted the privilege to snoop into a person's browsing activity. There are many different personal assistants, such as those described in the examples above, and while they all provide different services, they all share basic functionality. This thesis introduces an application framework as a technical approach in building and deploying these assistants. This application framework provides a generic and reusable architecture, design, and implementation method for constructing personal assistants.

The thesis presents the requirements, design, and important elements of the implementation of ***Sandwich*** – an application framework that contains a set of extensible classes that provide the basic services used by these personal assistants. In demonstrating ***Sandwich***, two personal assistants are prototyped. The hook model is deployed in documenting how to use ***Sandwich***.

1.2. Thesis Roadmap

We begin each chapter with an overview of the chapter's focus, followed by an outline of the chapter's sections. Chapter 2 covers background information, related work, and the current state of the art in adding assistantship into today's World Wide Web (WWW) architecture. This chapter provides the context for this thesis. Chapter 3 introduces and justifies the technical solution explored by this thesis - object-oriented (OO) application frameworks and the hook model. Chapter 4 discusses ***Sandwich***, in the context of its underlying architecture, design, design rationale, and extensibility through the use of the hook model. Chapter 5 evaluates on the hook model and the Unified Modeling Language (UML) notation standard. Both of these are used in documenting the framework. Chapter 5 also includes a detailed comparison between ***Sandwich*** and a closely related work, Webby. Finally, Chapter 6 concludes and outlines future directions.

2. Background, Related Work and Thesis Approach

When the WWW became generally known in 1994, the architecture deployed was relatively simple: the Mosaic or Netscape web clients simply made a request for resources (e.g. text files, GIF images and executables) to the desired web server. The web server then responded with the resources, if available. This communication was carried out with the HTTP protocol, specified by the World Wide Web Consortium [W3C]. As the Internet continued to grow, the number of web servers and clients increased. The average number of pages visited by an active web surfer increased dramatically. There has been on going effort to add application-specific capabilities through some form of assistantship to the web architecture.

Section 2.1 and 2.2 discuss two categories of assistantship: system-wide versus personal. A system-wide assistant focuses on a group of people, often belonging to the same Intranet; a personal assistant focuses on helping an individual in a certain way. Section 2.3 described another related domain area: agent systems. The similarities and differences between personal assistantship and this area of study are reviewed. This section closes with an elaboration on why the word "assistant" is used rather than "agent" and a working definition for "assistant" used in this thesis. The framework developed in this thesis relies on the HTTP protocol. Section 2.4 is dedicated to providing some HTTP background.

2.1. System Wide Assistants

Figure 2 shows the current primitive web architecture, which is inadequate in many ways as the web continues to grow. A common shortcoming is that a corporation is unable to restrict its access availability to selected parties. Figure 3 shows an enhanced version. Here, system-wide assistant is added through the use of a system-wide proxy. Specifically, the kind of assistantship in this example is a proxy that plays the role of a firewall.

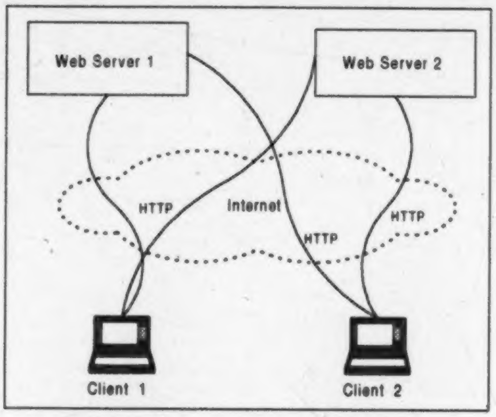

Web Server 1
Web Server 2
HTTP
Internet
HTTP
HTTP
Client 1
Client 2

Figure 2: A Primitive Web Infrastructure

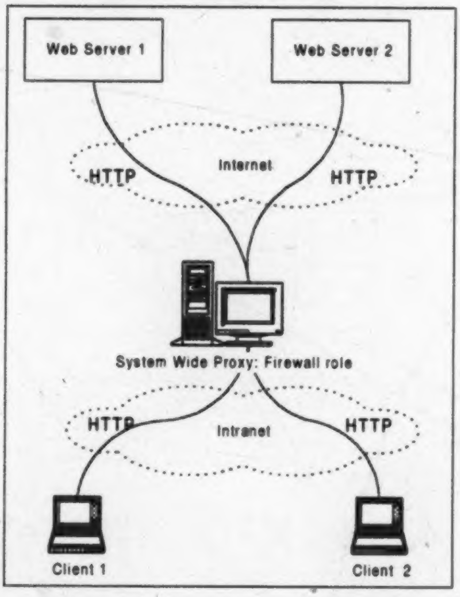

Web Server 1
Web Server 2
HTTP
Internet
HTTP
System Wide Proxy: Firewall role
HTTP
Intranet
HTTP
Client 1
Client 2

Figure 3: Web Infrastructure That Deploys System Wide Proxy

There are many different forms of proxies, but usually they are categorized according to which level they operate at [Luotonen98]. Three levels of proxies are commonly identified and briefly described below. Proxies from any of the three levels can be used to provide assistantship. Currently, system-wide proxies are already widely deployed by major firms. Table 1 gives a summary of these three levels along with some common examples that give system-wide assistantship.

Three Levels	**Meaning**	**Examples**
Packet Level	Makes use of TCP/IP packet information.	Router
Connection (Circuit) Level	Makes use of connection establishment information, such as port numbers.	Socks (firewall role)
Application Level	Makes use of application protocol knowledge	Internet caching and load-balancing software as well as ***Sandwich***.

Table 1: Proxy Categories and Examples

1. **Packet Level**: Proxies at this level use packet information to perform their tasks. An example is a router that performs simple packet filtering based on the TCP/IP header data in the network packets.

2. **Connection (Circuit) Level**: Proxies belonging to this level are software programs that act at the connection level, i.e. during connection establishment and are often based on port numbers. A proxy operating at this level has no prior knowledge of the communication protocol. It simply forwards data in both directions in the connection. The most widely used example here is SOCKS [Socks]. SOCKS enables hosts on one side of the SOCKS server to gain full access to hosts on the other side of the SOCKS server without requiring direct IP reachability. Thus, a SOCKS server is often used in supporting network firewalls, enabling hosts behind a SOCKS server to gain full access to the Internet, while preventing unauthorized access from the Internet to the internal hosts.

3. **Application Level**: A proxy at this level understands certain application protocols, such as HTTP, NNTP and FTP and uses this knowledge to accomplish its tasks. Every application proxy server has some embedded functionality. For example, a load-balancing proxy dynamically allocates requests to different servers based on the overall load of the system, ensuring that loads are distributed equally. A caching proxy such as Squid Internet Object Cache [Squid] and CacheFlow [CacheFlow]

specializes in caching resources that it captures in order to reduce network congestion or improve speed.

2.2. Personal Assistants

Personal assistants attempt to provide assistantship to an individual rather than a group as do system-wide assistants. With respect to the widely deployed web architecture shown in Figure 2, personal assistantship can be added on (a) the client side, with web browsers (b) the server side, or (c) in-between. Each of these will be briefly described with respect to current state-of-the-art technologies deployed in adding personal assistantship. We conclude by indicating that ***Sandwich*** uses the in-between approach.

2.2.1 Client Side

On the client side, plug-in technology is the most mature method of implementing personal assistants. Some examples of popular multimedia plug-ins are RealPlayer, QuickTime, and Shockwave. Another emerging method is on Extensible Markup Language (XML); W3C is in the process of defining this that can be presented with Extensible Style Language (XSL) and/or Cascading Style Sheets (CSS).

2.2.2 Server Side

On the server side, many extensions to web servers are already in use in providing personalized page that may assist the user. For example, most of today's non-static pages or forms are generated dynamically. The most popular and widely deployed web server extension on the web server to allow such dynamic generation is the Common Gateway Interface (CGI) script, which can be written in any language, Perl being the most popular. The main drawback of this approach is that each time a request is made to a CGI script, a new process is started to interpret the script that manipulates the request. Java Servlet [Sun, Hunter98] is the other more recently supported web server extension that is gaining popularity. A servlet provides the same capabilities as a CGI script without the aforementioned drawback. A servlet is initialized and loaded to the Java Virtual Machine (JVM) only once, and thus the overhead of starting a new process is eliminated.

2.2.3 In-Between

"In-between" proxies provide system-wide assistantship through firewalls, caching and traffic monitoring. Because most of today's proxies are system-wide assistants that are designed to act on behalf of the servers, they are often placed close to the server, i.e. the proxy server is installed on the same machine as the server or another machine in the same Intranet as the server. There has been a limited amount of research in using a personal proxy that acts on behalf of an individual (i.e. proxies playing the role of a personal assistant).

2.2.4 Approach Summary

This thesis investigates the third or the in-between approach, i.e. the use of a proxy to provide personal web assistantship. This approach has the advantage over the client side and server side approaches because it is independent of both the browser and the web server. We consider the simplest architecture as illustrated in **Figure 4**.

According to the proxy classification given previously, the proxy server developed in this thesis is an application level one that understands the HTTP protocol used on the WWW. Other protocol examples are FTP, Gopher and NNTP. Using this form of proxy as the basis, we develop an application framework for personal web assistants - ***Sandwich***. ***Sandwich*** provides extensible areas that allow users to construct and attach their desired web assistants.

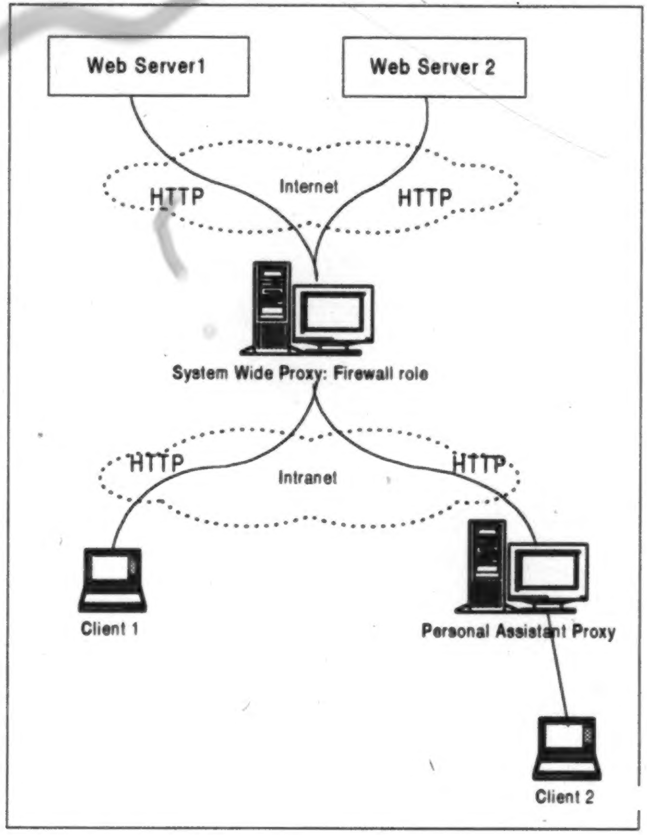

Web Server1
Web Server 2
HTTP
Internet
HTTP
System Wide Proxy: Firewall role
HTTP
Intranet
HTTP
Client 1
Personal Assistant Proxy
Client 2

Figure 4: ***Sandwich*** Infrastructure

2.3. Agent Systems

The word "agent" has come to mean many things in the area of information technology. We include a discussion of agents in this thesis to address questions about the similarities and differences of agents and assistants in this thesis. We first provide the background of related work by (1) quoting from a commonly referred to paper on agent technology, (2) categorizing agents based on their functions, and (3) summarizing several major projects on agent systems. We conclude this section with the similarities and differences between different focuses of agents and this work.

2.3.1 Agent Definition

Agent technology is becoming a popular and important research topic and is being applied in diverse domains in different communities, such as artificial intelligence (AI), distributed computing, networking and software engineering. Each community is interested in slightly different aspects of agents and the word "agent" is being used widely in many articles without a real definition. A commonly referenced paper entitled "Is it an Agent, or just a Program? A Taxonomy for Autonomous Agents" [FG96] contains a widely accepted definition of an autonomous agent:

> *"An **autonomous agent** is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future."*

This paper also makes a distinction between software programs and agents. All software agents are programs, but the converse is not true; an ordinary program that simply takes its input, acts on it and gives an output is not an agent. The paper breaks down autonomous agents into different types; a software agent is an immediate type of autonomous agent. The paper, unfortunately, leaves as an open issue on further sub-classification of software agents.

2.3.2 Agent Categories

For simplicity, we have classified software agents by their functions. The following is a brief description of popular agent types appearing in current agent-related literature.

- **Intelligent agents** are probably the most primitive type of agents on which AI communities have conducted research. This type of agents contains AI algorithms for reasoning, planning, and learning in order to adapt to an environment and solve a certain set of problems in a changing environment.

- **Internet agents** are one of the most widely studied phenomena today due to the rapid growth and interest in the Internet. Information filtering agents, search engines, and negotiating agents are examples of this category.

- **Communicative agents** are software systems that are able to communicate with other agents, whether they be a human or another software agent. There is still no agreement on a standard protocol for agent-to-agent communication. Some of the existing communicative agents have no support for standard protocol for agent-to-agent communication, they simply use direct method calls as the technique. Other more sophisticated agents use Knowledge Query Manipulating Language (KQML) as the Agent Communication Language (ACL).

- **Mobile agents** are able to transport themselves from one machine to another. This new type of agent has arisen because of recent interest and emphasis on distributed technology such as CORBA from OMG, DCOM from Microsoft and Java RMI from Sun.

2.3.3 Agent System Projects

Table 2 summarizes major on-going agent-related projects. From the table, we see that most focus on mobile and communicative agent technology, with only a few directed towards the Internet or Intelligent agents.

Project Name	Institutions or Company	Emphasis
Ajanta [Ajanta]	University of Minnesota	Infrastructure for mobile agents: security and robustness.
Aglets [Aglets]	IBM	Mobile and multi-agent framework, use of Agent Transfer Protocol (ATP) for distributing and transferring agents.
Concordia [Concordia]	Mitsubishi	Mobile agents for network management (reliability and security)
D'Agent (Agent TCL) [DAgent]	Dartmouth College	Mobile agent systems for information retrieval, presentation, integration and distribution.
InfoSleuth [InfoSleuth]	MCC	A mixture of distributed mobile agents, KQML support, and knowledge-based management systems (CLIPS).
JACK Intelligent Agents [JACK]	Agent Oriented Software Pty. Ltd (Australia)	Framework to create agent-to-agent systems using component-based approach. Agents range from simple data retrieval to those that understand Belief Desire Intent (BDI).
JATLite [JATLite]	University of Stanford	Templates for creating agents. Agents communicate to each other using KQML.

Java-To-Go [JavaToGo]	UC Berkeley	Itinerative computing (site-to-site computations required by mobile agents).
Odyssey (Telescript) [Odyssey]	General Magic	Transportable agents that can perform certain task at a mobile host and return the results.
RESTINA [Restina]	CMU	Goal-directed agents using a task-centered approach.
TACOMA [Tacoma]	Tormso, Cornell and California University	Operating support for mobile agents, especially fault-tolerance aspects.
WebeW [WebeW]	Authentech Inc.	Helps users to effectively browse and search the web.
WebMate [WebMate]	CMU	Provides personalized web system.
Webby (WBI) [Webby]	IBM	Provides a flexible API for programming intermediaries known as plug-in on the Web.

Table 2: Agent Related Projects

Although most of the projects in Table 2 focus on mobile and a multi-agent system, each has a different emphasis on the software modalities (e.g. security, performance, fault tolerance, etc) imposed by distributed technologies, and each focuses on a different domain. We also see that only a few support intelligent agents.

The three most relevant projects to ***Sandwich*** are WebeW, WebMate and Webby, all of which are personal agent systems. Their commonalties with ***Sandwich*** lie in the following factors:

- They are targeted towards internet agents, according to the above categories,
- They are written in Java, and
- They use an HTTP application level proxy

WebeW and WebMate are different from ***Sandwich*** in that they both have assistantship built into their proxy already and their documentation does not indicate how the user can add in new assistants or remove an existing one. WebeW and WebMate might have been constructed from an application framework but the use of the framework is not visible to the user. Webby is closer to ***Sandwich***, and a detailed comparison with it is deferred to Section 5.3.

2.3.4 Assistant

The question why we have chosen to use the word "assistant" instead of "agent" needs to be answered. As we see from the above, agents cover a broad area of studies and carry many different definitions. One characteristic that people often associate with an "agent" is the ability to learn and to act accordingly to the knowledge acquired. This is not true for assistant. In addition, ***Sandwich***'s assistants are neither inherently intelligent nor mobile, and therefore the word "agent" seems inappropriate. The following working definition clearly describes the type of "assistant" that is used in this thesis.

> An *assistant* is a software program that supports personal use of the Internet via a browser. The two types of assistants that we focus in this thesis are observing and delegating assistant. An observing assistant monitors, measures, and reports on the user's browsed content and a delegating assistant monitors and acts on the user's behalf for postable pages.

2.4. HTTP

This thesis makes extensive use of the Hypertext Transfer Protocol (HTTP) application protocol that is widely used on today's WWW. Even though HTTP is a well-known protocol, its internals are not well understood by most people. The purpose of this section is to provide the necessary background knowledge on this protocol to understand the design of ***Sandwich***.

HTTP protocol is used as the communication protocol between web clients (i.e. browsers) and web servers. See Figure 5. In short, when the user requests resources such as images, text or applications, from a web server, the browser sends an HTTP request to the remote server or intermediate proxy, if such a proxy exists. The answer to the request is referred as the HTTP response.

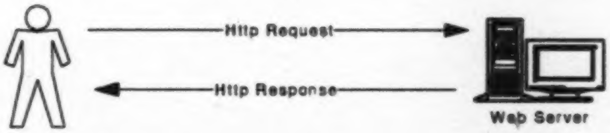


Figure 5: HTTP Request and Response

An **HTTP request** is comprised of several lines of string in the following order:

1. The first line gives the request method (GET, POST, etc), the Uniform Resource Locator (URL), and the browser's supported HTTP version.
2. Several lines of headers and their values, each line containing of a particular HTTP header name, followed by a colon and then the value,
3. An optional request body. This contains the name-value pairs for the POST method.

An **HTTP response** is comprised of several lines of string or binary data in the following order:

1. The first line gives the numeric response code, the string response message, and the web server supported HTTP version.
2. Several lines of headers and their values, each line containing a particular HTTP header name, followed by a colon and then the header value.
3. An optional response body that contains the HTTP content itself. This can be in ASCII or binary data. For example, an HTML page would be in ASCII and a GIF file would be in binary form.

Four identified types of **HTTP headers** are general, request, response and entity. General and entity types of headers are meaningful in any HTTP request or HTTP response. Request type headers are only meaningful in HTTP requests; response type headers are only meaningful in HTTP responses. Given the structure of HTTP, we will now analyze an actual example.

2.4.1 Example

The following is a basic HTTP request and response example from a user who wants to go to the home page of the University of Alberta, located at URL http://www.cs.ualberta.ca/. To initiate this request, the following command is sent from the user's web browser to the university's web server.

```
GET http://www.ualberta.ca/index.html HTTP/1.1<cr><lf>
Proxy-Connection:  Keep-Alive<cr><lf>
User-Agent:  Mozilla/4.05 [en] (WinNT; I ;Nav)<cr><lf>
Host:  www.ualberta.ca<cr><lf>
Accept: image/gif, image/x-xbitmap, image/jpeg, image/png, */*<cr><lf>
Accept-Language:  en<cr><lf>
Accept-Charset:  iso-8859-1,*,utf-8<cr><lf>
```

When the university's web server on www.ualberta.ca receives this command, it checks whether the requested resource (i.e. index.html) exists or not. If it does, then the web server returns an HTTP response with the OK status, such as the following.

```
200 OK HTTP/1.1 <cr><lf>
Date: Wed, 07 Apr 1999 21:32:04 GMT <cr><lf>
Server: Apache/1.3.3 (Unix) <cr><lf>
Last-Modified: Wed, 07 Apr 1999 21:22:36 GMT <cr><lf>
ETag: "14df9-102d-370bcc9c" <cr><lf>
Accept-Ranges: bytes <cr><lf>
Content-Length: 4141 <cr><lf>
Keep-Alive: timeout=15, max=100 <cr><lf>
Connection: Keep-Alive <cr><lf>
Content-Type: text/html <cr><lf>
<cr><lf>
(The content itself is here, i.e. the index.html of http://www.ualberta.ca/)
```

We will not explain each header in detail here. Appendix A lists alphabetically on all header fields defined by the HTTP 1.1 specification [W3C]. Note that <cr> stands for carriage return and <lf> stands for linefeed.

2.4.2 Approach Summary

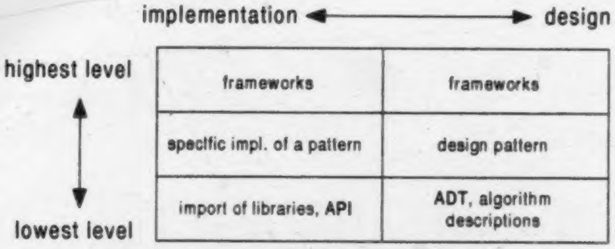
The underlying architecture of ***Sandwich*** is based on HTTP application-level proxy architecture. Since over 80% of Internet communications are accomplished through HTTP, it makes perfect sense to tackle the problem by having ***Sandwich*** understand the HTTP protocol. The other protocols that occupy the rest of the Internet traffic include NNTP for newsgroup, SMTP for e-mail, and FTP for file transfers.

3. Hooking into Application Frameworks

In this chapter, we cover the background information on the technical approach used in this thesis. Section 3.1 introduces and reviews the present research status of object technology, different forms of reuse and frameworks in general. Section 3.2 elaborates on application frameworks, the type of framework explored in this thesis. This section provides an overview on application frameworks and hooks, the differences between application frameworks and applications, the pros and cons of deploying application frameworks, and the documentation techniques for frameworks that exist today. Section 3.3 provides background information on the hook model proposed by Garry Froehlich [Froe96] and experimented with in this thesis.

3.1. Object Technology and Reuse

Software development approaches have changed significantly over the years. Their evolution has been motivated by the need to produce high quality software that is easy to maintain and that can evolve at a fast pace. Producing software that can easily be reused is a step towards this goal. Currently, object technology is the most promising way of improving the software development process and providing for software reuse that, in the long run, can reduce development time.



frameworks	frameworks
specific impl. of a pattern	design pattern
import of libraries, API	ADT, algorithm descriptions

Figure 6: Reuse Forms and Levels

There are many ways to accomplish software reuse and Figure 6 illustrates the major forms of software reuse and their levels of support. The evolution of software development has gradually shifted from only reusing code through Application Programming Interfaces (APIs), design reuse through patterns [GHJV95] to both design and code reuse through the use of frameworks. Before we proceed to the discussion on the current status of reuse practice using these major forms, we provide an overview the different framework types that exist today.

3.1.1 Types of Frameworks and Examples

There is no one particular way to classify the "type" of a given framework. Two popular means for classifying frameworks are summarized below.

1. Based on the nature of the domain of a framework, frameworks can be roughly classified in to the following two types.
   a) **Foundation frameworks** contain supports for functionality that can be applied in various domains. They often have a very flexible, interconnection mechanism for hooking application-specific code to the framework. Examples are the Graphical User Interface (GUI) frameworks such as Microsoft Foundation Class (MFC). [FS97] refers to this group of frameworks as the middle-ware integration frameworks.
   b) **Application frameworks** contain supports for a particular domain. They often have a restricted number of hot spots and more constraints in using them. Examples are Speech Recognition Framework [Srinivasan99], IBM San Francisco for the domain of E-Commerce (http://www.ibm.com/java/Sanfrancisco/), and ET++ Swaps Manager in the Financial Engineering Domain [EggGam92]. [FS97] refers to this group of frameworks as the enterprise application frameworks.

2. Based on the techniques used to extend frameworks, i.e. their adaption mechanisms, frameworks can be roughly categorized into the following two major (rather extreme) types.
   a) **White box frameworks** are extended through object-oriented (OO) language features such as inheritance and dynamic binding. [FS97] describe a white box framework as one that can be extended either by (a) inheriting an abstract class and/or (b) overriding pre-defined hook methods that are predefined by the framework when a pattern such as template [GHJV95] is deployed.
   b) **Black box frameworks** are extended via the plug and play approach, specifically via object composition or changing component properties. It resembles the notion of components and beans development.

   It is important to note that an application framework is often a mixture of these two extensions rather than only one approach.

3.1.2 Current State of Reuse Practice

The current state of reuse practice lies mostly on the lowest level (APIs), with design patterns [GHJV95] gaining popularity in the past couple of years. The maturity of reuse in the form of frameworks differs depending on the type of frameworks in discussion. Foundation frameworks are reasonably mature; for instance MFC is extensively deployed in today's on-going software development effort. On the other hand, application frameworks are a less mature reuse practice and are still emerging. There is a strong belief in the community that application frameworks are promising in terms of delivering significant levels of reuse by leveraging the existing design and implementation common to all application in the same domain. Thus, application frameworks provide reuse in many facets, from system architecture, design, and implementation to conceptual reuse that includes set of common terminology and view on the problem domain in discussion. Nonetheless, several research problems remain to be solved in this area. Some examples are the assessment or evaluation of frameworks, weak justification for using frameworks (due to steep learning curve), process for developing frameworks, integration of multiple frameworks, and documentation strategy for frameworks.

3.1.3 Approach Summary

Based on the nature of the domain, the framework proposed in this thesis work, ***Sandwich***, is an application framework that focuses on the domain of personal web assistants. It is important to note that other sub-frameworks, such as the persistence framework, that are used in ***Sandwich*** may still be considered as a foundation framework. Based on the adaption mechanism classification scheme, ***Sandwich*** is mainly a white box framework that deploys several black box components such as free third-party beans that are downloaded.

Among the listed outstanding research problems for frameworks at the end of Section 3.1.2, this thesis provides a case study for using the hook model as the strategy in documenting frameworks.

3.2. Application Frameworks

An object-oriented application framework is a set of classes and their collaborations that provide a generic and reusable architecture, design, and implementation method for a given family of applications. An application framework consists of extensible area known as hot spots. To create an application from a framework, one or more hooks are enacted to generate application code that is attached to one or more hot spots of the framework. See Figure 7. From here onwards, the use of the word "framework" implies application framework.

A framework sometimes comes with a set of default application code already generated by enacting some of the hooks for some of the hot spots. Framework users can replace these default features with their desired one easily, simply by re-enacting the set of hooks based on their needs. Frozen spots of a framework are areas where framework implementations are already firmly in place. Developers are strongly encouraged not to change these frozen spots, or otherwise, be involved in the framework evolution.

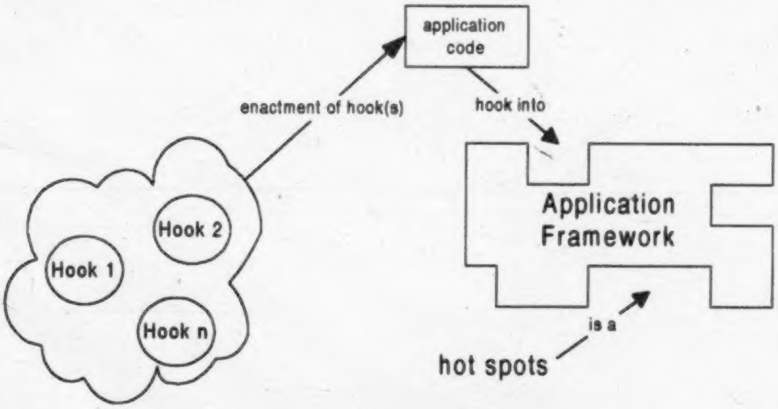


Figure 7: Hooks and Hot Spots

3.2.1 Application Frameworks versus Applications

An application framework is different from an application. An application framework can be used to develop multiple applications belonging to the same domain. An application, on the other hand, can be developed from scratch or from a framework. In the latter case, the application is said to be an instance of the framework that it is constructed from. The two aspects of their differences are [FHLS98b]:

(1) Level of abstraction. Frameworks are more abstract than applications. The design of a framework must be flexible and abstract enough to support a family of applications, and

(2) Frameworks are, by their nature, incomplete; applications are typically self-contained and thus complete by themselves.

A framework carries all the benefits of developing OO applications. In the context of reuse, frameworks promise to deliver the benefits to a greater extent. A typical application reuse is often in the form of implementation such as using some third party APIs. An application framework provides for code reuse as well as design and domain-specific reuse such as common terminology and view of the problem domain.

Due to the differences between frameworks and applications, the methodology in developing frameworks is also different from that used in developing object-oriented applications. The most important difference is the notion of inverted control. When developing an OO application, developers have full control over the main control loop of the application. This is not true when developing an application framework or creating an application from an application framework. By its very nature, much of the design for an application is already completed before development commences when using an application framework. Therefore a developer must be willing to reuse someone else's code and design. As an example, the developer may not be able to alter the application's main control because it is directed by the framework's embedded flow of control. This is also known as the "Hollywood Principle" or the "Don't call us, we'll call you" approach. A generally accepted principle in framework use is that a framework developer has to make sure that the flow of control embedded in the framework can support the domain of applications targeted by the framework.

3.2.2 The Pros and Cons of Application Frameworks

The decision to build any type of framework requires the consideration of many issues, most importantly is whether or not a framework is needed. Although frameworks are promising in terms of delivering a high degree of software reuse, there are also drawbacks in using it. In this section, we summarize the pros and cons in building application frameworks.

Pros:

1. **Reuse.** As we have seen in Section 3.1, frameworks provide not only the code reuse but also design and architecture.
2. **Maintenance.** All applications constructed from the framework share the common design and code base and thus make the maintenance of these applications easier.
3. **Quality.** In addition to providing design reuse, the framework is also a tested design proven to work and thus forming a quality base for developing new applications [FHLS99b]

Cons:

1. **Steep learning curve.** There is often a steep learning curve for developers who need to familiarize themselves with the frameworks chosen for building applications from. Often, some level of object knowledge is required for framework users who are not the original framework developers to use a framework effectively.

2. **Knowledge Transfer Challenges**. Framework developers often face challenges when the knowledge accrued in constructing the framework needs to be transferred to other developers or maintainers.
3. **Integration Challenges.** Two main difficulties can be encountered when multiple frameworks are involved in creating an application. [FHLS99c] describes these challenges further from the hook perspective.
    (a) A framework gap when services expected by a framework are not available from any other frameworks deployed in the system, and
    (b) A framework overlap when two or more frameworks provide the same service through different means.
4. **Scope**. The breath of a framework is hard to determine when designing a framework. The broader the area a framework covers, the less focus it has. On the other hand, if the framework is too focused then its reuse may be limited.
5. **Design Authority.** To effectively use a framework, the application developers must yield design authority. For example, their applications must conform to the overall architecture and control loop of the framework chosen. Recall the "Hollywood Principle" described in Section 3.2.1.

3.2.3 Framework Documentation

A good set of documentation is indispensable in order for frameworks to deliver their promises and potential. Good documentation helps in various ways. For the framework users, good documentation eases the process of hooking in new applications; for the framework developers, documentation promotes communication, understanding of framework's dynamics and internal design, and finally for framework maintainers, documentation helps them to comprehend objects, modules, and subsystem dependencies.

But what constitutes a good set of documentation for frameworks is still an open issue. There have been several framework documentation techniques proposed such as patterns from [Johnson92], a cookbook from [KP88], formal specifications contract from [HHG90], exemplars from [GM95], motifs from [LK94] and hooks from [Froe96]. In this thesis, the hook model proposed by [Froe96] is experimented with and used as the documentation technique for ***Sandwich***. Experience based on this method will be collected and summarized. The following section reviews some background knowledge on the hook model.

3.3. The Hook Model

An important aspect of a framework's classes is that they are adaptable; more specifically a framework provides some mechanisms by which its classes can be extended. Sub-classing and registering callbacks are examples of hooks. These extensions need to be well documented so that developers who use the framework to develop an application know exactly where to look for help. The hook model proposed by Garry Froehlich is a documentation technique that strives to provide an answer to this. A typical hook for a framework is organized and written using the following template [FHLS98a].

Name	A unique name, within the context of the framework, that identifies the hook
Requirement	A textual description of the problem the hook is intended to help solve. The framework builder anticipates the requirements that an application will have and describes hooks for those requirements
Type[1]	An ordered pair consisting of the method of adaption and the amount of support provided for the problem within the framework
Area	The parts of the framework that are affected by the hook
Uses	The other hooks required for using this hook. The use of a single hook may not be enough to completely fulfill a requirement that has several aspects to it, so this section states the other hooks that are needed to help fulfill the requirement
Participants	The components that participate in the hook. There are both existing and new components
Preconditions[2]	Constraints that must be true before applying the hook.
Changes	The main section of the hook that outlines the changes to the interfaces, associations, and control flow amongst the components given in the participants section
Post-conditions[2]	Constraints that must be true after applying the hook.
Comments	Any additional description needed.

Table 3: Hook Template

---

[1] These ordered pairs are derived from the hook types that are elaborated in the following paragraphs.

[2] Older version of the hook model included a field known as "Constraints". This has been replaced by "Preconditions" and "Post-conditions".

The type of hooks is determined using a two dimensional system: the method of adaption and the level of support. The method of adaption dimension is based on how the framework is affected (method of adapting) when the hook is applied; in particular whether the framework's features are being enabled, disabled, replaced, augmented or added. The level of support dimension is the opaqueness of the hooks.

Under the method of adaption dimension, we have five types of hooks:

(a) **Enabling a Feature**. When hooks of this type are used, a particular feature of the framework (that is not on by default) is turned on, often through a black box approach.

(b) **Disabling a Feature**. This is similar to the above type, except here the feature of the framework is turned off and may involve the disruption of other features.

(c) **Replacing a Feature**. When hooks of this type are used, a particular feature of the framework is being replaced with a new feature that satisfies the same interface and behavioral obligations as before.

(d) **Augmenting a Feature**. When hooks of this type are used, the existing flow of control of the framework is intercepted with additional needed actions and then control is returned back to the framework. Unlike replacing a feature, augmenting a feature adds to the behavior without redefining the feature.

(e) **Adding a Feature**. When hooks of this type are used, a new capability that did not exist before is added to the framework.

Under the level of support dimension, we have three types of hooks:

(a) **Option**. Hooks here are the easiest to use; pre-built components are often attached to the framework using the plug and play approach. However, they are less flexible. Often, a black box framework contains more hooks at this level than any other levels.

(b) **Pattern**. Hooks here are more flexible than those at the option level. Supplying parameters to a class and sub-classing are the commonly used hook techniques. Steps required to enact the hook(s) in fulfilling a requirement are outlined but details of the codes are deferred as they depend on the application being developed. Often, a white box framework contains more hooks at this level. When more than one class is involved in the hook, the level of support is also known as collaboration pattern.

(c) **Open-Ended.** Hooks here are the most flexible ones and are more likely to be used by users who are evolving and extending the framework itself. A deeper understanding of the framework's internals is required.

The value of the "Type" field for a given hook is the method of adaption type followed by the level of support type. Some examples are enabling patterns, adding options, and so on.

Sandwich will use the hook model as the technique to describe its extensions. This experience will further evaluate the hook model developed by Garry Froehlich in his Ph.D. dissertation.

4. Sandwich

This chapter focuses on the architecture, design, prototypes and hooks of ***Sandwich***. Section 4.1 gives an overview on ***Sandwich***. Section 4.2 describes the actors involved in ***Sandwich***. Section 4.3 discusses ***Sandwich*** requirements that have been captured with use cases. Section 4.4 details ***Sandwich*** in terms of its subsystem architecture, design and its rationale, and hooks. Section 4.5 outlines the detail steps in creating new assistant in ***Sandwich*** and finally in Section 4.6 discusses other considerations such as assistant that plays two roles and assistant-to-assistant collaboration.

4.1. Overview

The name ***Sandwich*** was chosen mainly because of the underlying architecture of the framework. ***Sandwich*** is essentially built from the notion of a proxy server, an intermediary that lies between a server and its clients, or is "sandwiched".

As the WWW continues to grow, the question of what is the best strategy for an Internet user to manage and control the pages that he or she has visited or wants to visit becomes more and more important. Currently, there is no universal solution to this problem of data management. Nonetheless, an attractive and viable answer would appear to involve personal assistantship add-ins. We intend to explore this in this thesis.

Many assistant programs have been identified and built; however, almost all of them solve a particular problem in a separate application domain. Also, each of these encompasses their own architecture, design, and set of codes. However, because of their commonality, many of these applications could share the same architecture and design in a proxy. For example, [SY97] also tried to achieve additional or specific application functionality through the use of an intermediate proxy. It is the goal of this thesis to identify these common services, and to design and develop the corresponding building blocks. An application solves a particular problem; a framework provides the solution basis to a set of common problems. Assistant programs are customized, extended and built from the ***Sandwich*** from the documented hooks. ***Sandwich*** is intended to be small and very focused.

Based on the techniques used by assistants to accomplish their objectives, we have identified two groups of assistants. The first group is comprised of standalone assistants that are independent of the user's browsing activities. The second group is comprised of assistants that are dependent on the user's browsing activities and can be further broken down into two types: observing and delegating assistants, based on their interactions with the end-users. The following figure depicts this taxonomy.

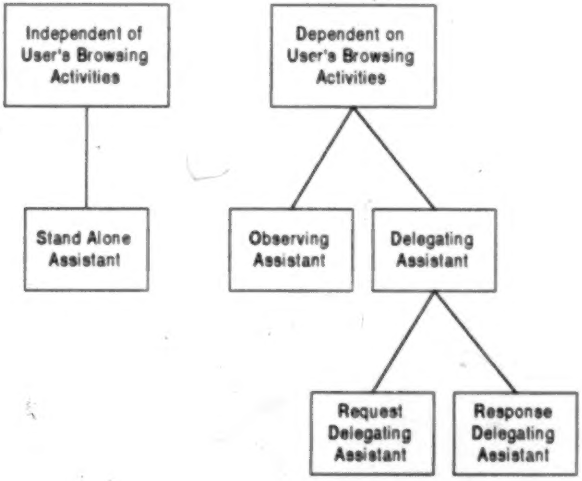

Independent of User's Browsing Activities
Dependent on User's Browsing Activities
Stand Alone Assistant
Observing Assistant
Delegating Assistant
Request Delegating Assistant
Response Delegating Assistant

Figure 8: Groups and Types of Assistants

This thesis explores a framework that supports the construction of the second group of assistants, i.e. those that depend on the user's browsing activities. Thus, only observing and delegating assistants are explored and elaborated on in this thesis.

Although observing and delegating assistants belong to the same group, they are slightly different. The services provided by observing assistants can be accomplished by watching the user's web activities and involve almost no dynamic user interaction during this observation process. The output of this observation is often a meaningful output, in the form of a report. The end user explicitly asks the assistant to present this meaningful output. The services provided by delegating assistants are more dynamic and involve a lot more user interaction. A delegate monitors and detects the right time for delegation. When an opportunity exists, it will prompt the user for confirmation before proceeding with delegation. If the user agrees, then the delegation process continues. Otherwise, the delegation is aborted.

4.2. *Sandwich* Actors

Actor is a term used in the UML [RGB99, Fowler97] requirement analysis phase for describing a system's end-user role. Like any other software system, ***Sandwich*** has many actors at different levels of usage. [FHLS99b] describes three different roles (framework designers, framework users, and framework maintainers) associated with the development and use of an application framework. All these roles as well as an additional one, end-user role, apply to ***Sandwich***. Each is briefly described in this section.

Different individuals do not necessarily fill these different roles. For instance, the framework designers may also play the role of framework maintainers. Also, the end-users may actually download the ***Sandwich*** source code and start developing assistants (i.e. play dual roles of assistant developer and end-user).

4.2.1 Framework Designers/Developers

These actors develop the original framework - ***Sandwich***. They possess the broadest knowledge of the internal structure and underlying architecture of ***Sandwich***. They define and write the hooks for extending the framework.

4.2.2 Framework Users (Assistant Developers)

These actors identify new assistant programs and develop them through the hooks defined and documented by the framework designers. By doing so, large portions of the framework architecture, design and implementation are being reused.

4.2.3 Framework Maintainers

These actors evolve and refine ***Sandwich***. For example, they would ensure that ***Sandwich*** is using the latest HTTP specification recognized by W3C.

4.2.4 End Users

These actors use ***Sandwich*** to facilitate and have more control over their daily web surfing capabilities. When they first use ***Sandwich***, they download, install and use ***Sandwich*** "as is" (***Sandwich*** packaging comes with a pool of personal assistants). Later, these end users may purchase or even develop new personal assistants and add them into ***Sandwich***. They may also assist in defining new assistant programs together with assistant developers.

4.3. Use Cases

In this section, we will first provide an introduction to "use case," including its origin, a description of the current state-of-the-art and definitions of the terminology involved. We then summarize how use cases are applied during the analysis phase of ***Sandwich***. Use cases are used to capture requirements for the family of applications that ***Sandwich*** targets (i.e. a few assistant examples) and to generalize requirements for the framework itself.

4.3.1 Origin and the Current State-of-the-Art

Ivar Jacobson first introduced the concept of use cases in object-oriented software engineering (OOSE) methodology. The basic idea is that the functionality of a software system can be captured as a number of different use cases. Use case has gained popularity since then and has become the standard in UML for capturing user requirements. There is common agreement among software engineers that use cases are useful. However, people are still very confused about the protocol in written use cases, in particular what to include and how to structure them.

4.3.2 Introduction

A **use case**, denoted as an oval, is a collection of possible sequences of interactions between the system under discussion and its external actor(s). Figure 9 below is an example of a use case diagram. An **actor** represents a role that entities (someone or something) in the external environment can play in relation to the system. An actor is denoted as a person, however, the actor is not the actual person but rather the role the person or thing plays. The relationship between an actor and a use case is denoted with an arrow.

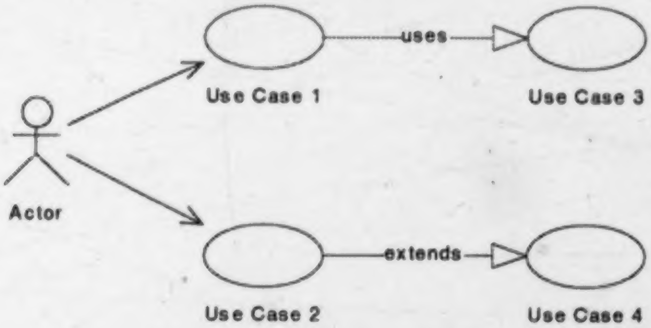


Figure 9: Example of a Use Case Diagram

In addition to the links between actors and use cases, there are two other types of links: uses and extends. Both represent the relationship between two use cases and is denoted with an arrow and a text label (uses or extend). They imply factoring out common behavior from multiple use cases to a single use case; however, their intents are different. An **extend** relationship is used when one use case is similar to another use case but does a little bit more. The rule of thumb to use **extend** is to do so for cases which are variations on normal behavior. A **uses** relationship occurs when a particular behavior is similar across more than one use case and thus can be factored out and be reused by all the applicable use cases.

A **scenario**, sometimes referred to as a transaction, is a term commonly confused with use case and is being used inconsistently. For instance, a scenario sometimes is used as a synonym for use case. In UML and this thesis, a **scenario** refers to a single path through a use case, that is, like a use case instance. Thus, a particular use case can be **realized** in one or more scenarios or has many realizations.

4.3.3 Applying Use Cases

Applying use cases to framework development is a little different than for application development. When use cases are applied to framework development, the use cases analysis becomes complicated, because of the differences between a framework and an application as discussed in Section 3.2.1. Recall that a framework is the skeleton that allows multiple applications in a certain domain to be built from. Thus, the use cases of a framework are not targeted towards one application but multiple related applications, often referred to as a family of applications. In contrast, when use cases are applied to application development, the use cases are simply the set of capabilities that the system needs to support to fulfill the one application's requirements.

We tackle this challenge by applying use cases to framework development in two stages:

1. Applying use cases to the ultimate family of applications that the framework is built for, and
2. Applying use cases to the framework itself.

Although they can be conceptually viewed as two distinct steps, they are highly correlated. For ***Sandwich***, during the analysis of the use cases of the family of applications, we capture and deduce the requirement patterns that would apply to group of personal assistants. These patterns allow us to factor out the common, reusable and extensible use cases shared by one or more assistants. These use cases are, in essence, the use cases of the framework. They are the basic services that ***Sandwich*** should provide to support its targeted family of personal assistant applications. This then leads to the reapplication of use case analysis, but this time to the framework itself.

4.3.4 Application Family Use Cases

In this section, use cases is presented for a representative family of personal assistant application. The following assistants were chosen for analysis purpose:

- Auto POST Assistant
- Search Summary Assistant
- PICS Filtering Assistant
- Simple Filtering Assistant
- Change Monitor Assistant
- Defer/Batch POST Assistant
- Enhanced History Assistant
- Time Keeper
- RFA Mailto Assistant
- Image Downloader

Each of the above assistants is described using the following template:

- As necessary, a background section that provides the knowledge to understand the assistant itself.
- Actor(s) involved
- Brief description of what the objectives and benefits of the assistant are
- Pre-conditions of the assistant in accomplishing its objective
- Main flow of the assistant in accomplishing its objective
- Additional information

4.3.4.1 Auto POST Assistant

Actor: End user

Description: An auto POST assistant detects an opportunity to automatically perform an HTTP POST using values that a user has posted before. This assistant can save a user significant time in entering the same POST values over and over again. A specific example is the stock quote obtainer example described in the introduction chapter.

Pre-conditions:
None

Main Flow:

1. The assistant remembers all POST requests: requested URLs and the posted name-value pairs.
2. The assistant monitors and detects delegation opportunity based on previously visited URLs, i.e. remembered in the above step.
3. Assistant prompts the actor whether or not to proceed with the delegation and lets the actor picks from a list containing previously posted name-value pairs, sorted according to the date the requests were made.
4. If the actor decides to go with a previously posted set of name-value pairs, this assistant generates a new request based on the one the user has picked from the list explicitly. The actor may choose to ignore the list and create a new set of name-value pairs.

Additional Information: None

4.3.4.2 Search Summary Assistant

Actor: End user

Description:

When the actor performs a search on a search engine, say Yahoo, this assistant will ask the user whether the same search is to be performed on the other popular search engines. If the actor agrees, the assistant gives a summary of all the URLs returned on the first page of all the search engines' results. This assistant allows web searching on different search engines concurrently and presents the search results in a consistent and perhaps personalized manner.

Pre-Conditions:

- The assistant contains a list of popular search engines URLs and allows the user to add/remove them.

Main Flow:

1. The assistant monitors and detects the delegation opportunity, specifically when the actor-requested URL is one of the above search-engine URLs that the assistant is aware of (see pre-conditions).
2. The assistant asks the actor whether or not to proceed with the delegation and lets the actor pick from the list of search-engine URLs.
3. If the actor decides to use the search summary assistant, this assistant will gather the search results from all the selected search engines. The assistant will then integrate the information and present it to the actor in a consistent manner. Even better, this assistant can allow the actor to specify the template for constructing the summary of all the search results.

Additional Information: WebCrawler is an existing site that provides such functionality with the exception that the user is not able to personalize the return pages.

4.3.4.3 PICS Filtering Assistant

Background:

PICS stands for Platform for Internet Content Selection and is an Internet ratings standard designed by W3C to allow Internet content selections. The idea is basically that PICS gives web publishers a standard way to describe the content of web sites or web pages. In short,

- The web site will be registered with a labeling scheme, such as Recreational Software Advisory Council on the Internet (RSACi at http://www.rsac.org/homepage.asp) and SafeSurf at http://www.safesurf.com/classify/index.html.

- All web pages of the web site will be attached and associated with labels, and labels contain a set of ratings.

- Labels can be generated either by the author of the web pages or through third party rating systems.

- Labels can reside in the head section of the web pages using HTTP META tag (see Appendix B), or within a label database or bureau accessible through the Internet.

- In essence, the label notion introduced here classifies Internet information and thus provides a scheme that allows content filtering/selection. Clearly, this infrastructure provides capabilities that are far beyond and on a larger scale than simple filtering based on a list of URLs. However, this system requires that all web pages be rated. [W3C] contains link to the full IETF specification for PICS.

PICS Filtering Assistant (Con't)

Actor: End user

Description:

This assistant allows the actor to perform content selection based on the ratings tagged to it using the PICS infrastructure. With the labels and ratings notion, this assistant ensures that only suitable material is presented to the actor. Inappropriate materials can be easily screened out for children.

Pre-conditions:

- The actor has specified which ratings are not allowed.
- The assistant is interested in knowing the META headers of all pages requested as well as the full requested URL.

Main Flow:

1. The assistant monitors and detects delegation, specifically if the response to the actor's request contains ratings that are included in the actor's disallowed ratings (see pre-conditions).
2. If so (i.e. a response contains rating included in the actor's disallowed ratings), then this assistant responds that the content is inappropriate to be viewed. Otherwise, the request is simply relayed.

Additional Information: [SY97] describes a technical solution (similar to ***Sandwich***'s approach) that will support this particular assistant.

4.3.4.4 Simple Filtering Assistant

Actor: End user

Description:

This assistant is the same as the PICS filtering assistant with the exception that it does not use the PICS infrastructure. This assistant allows the actor to perform content selection based on a list of URLs or wildcard URLs such as http://*.xxx.*. Like the PICS filtering assistant, this assistant ensures that only suitable material is presented to the actor. Inappropriate materials can be easily screened out for children.

Pre-conditions:

- The actor informs the assistant of the list of disallowed URLs.

Main Flow:

1. The assistant monitors and detects the delegation opportunity, specifically when an actor requests a URL that is one of the disallowed URLs (see pre-conditions).
2. When the opportunity exists, without prompting the actor, this assistant responds that the content is inappropriate to be viewed. Otherwise, the request is simply relayed.

Additional Information: None

4.3.4.5 Change Monitor Assistant

Actor: End user

Description:

Upon visiting a page that has been visited before, this assistant compares the previous page with the fresh pull page (from the remote web server). If the pages have changed, this assistant responds with a different page that shows the differences between the previously visited and the new page. This assistant can tell the actor right away whether or not a favorite page has changed since the last visit. It can also show the differences to the actor.

Pre-conditions:

- The assistant saves all pages that the actor visits. In order to limit the volume, the user might want to set a threshold on the number of pages or the amount of memory that can saved for this purpose. Alternative, the user can indicate to the assistant that only the top 10 favorite sites of his/her are to be saved for this purpose.

Main Flow:

1. When the actor revisits one of the pages that was saved according to the pre-condition, this assistant would prompt the actor if the actor wants to view the differences report.
2. If the actor agrees, then this assistant will go and fetch the new page, compare them and renders the difference report.
3. If the actor declines, then this assistant will simply relay the original request.

Additional Information: None

4.3.4.6 Defer/Batch POST Assistant

Actor: End user

Description:

This assistant allows an actor to defer a POST or batching up POST requests to remote web server(s). This is accomplished by having the capability to let the actor performs POST to the assistant only. This will defer the POST and thus batch POST becomes possible. Batch POST allows an actor to perform POST to multiple sites concurrently and to manage transactions with an all or nothing approach. For example, when making reservation for airline ticket and hotels, an actor might want to hang on to the hotel reservation until an airline ticket for the trip is confirmed. Currently, an actor has to open up multiple browsers to accomplish this.

Pre-conditions:

- The assistant monitors all POST requests by the actor.

Main Flow:

1. For each POST request, this assistant will ask the actor whether the POST should be deferred.
2. If the actor agrees, then the assistant saves the POST request parameters. The assistant then informs the actor that the POST request has been saved as a deferred request.
3. If the actor disapproves, this assistant simply relays the original request.
4. This assistant provides a separate interface that allows the actor to perform the batch POST request.

Additional Information:

- Implementation of this assistant will require further in-depth investigation. For example, current a web transaction cannot undo itself, what is the strategy of this assistant. The implementation of this assistant will depend on how the server-side components manage end user sessions. In addition, this assistant might only be feasible when all web transactions conform to a standard.

4.3.4.7 *Enhanced History Assistant*

Actor: End user

Description:

This assistant saves all the URLs of pages (along with statistics such as how pages are obtained and the number of failed requests and so on) that the actor has visited and allows the actor to navigate to them easily. This assistant gives an enhanced version of current browser's history list. It is an enhanced version since this assistant now remembers all URLs visited (on all sessions) persistently, unlike a browser history list that remembers only a limited number of URLs.

Pre-conditions:

- The assistant monitors all URLs that a user has visited.

Main Flow:

1. The assistant provides an interface that presents to the actor URLs of all pages visited (along with certain statistics) and allows the actor to go to the page again easily.

Additional Information: None

4.3.4.8 Time Keeper

Actor: End user

Description:

This assistant keeps track of the actor's on-line elapsed time so that an alert is triggered when an actor's on-line (assuming the user only uses a browser when online) time has reached a certain threshold. Some Internet service providers (ISPs) today still impose a limited amount of web-time for a certain discount price. When this limited time is used up, the actor will be charged a higher price. We see that a time keeper assistant behaves like a reminder and can be very handy in this respect.

Pre-conditions:

- The assistant needs to know the start and end timestamp of the user being on-line and off-line.
- The assistant allows the actor to set the threshold that triggers the alert. This threshold is the limited minutes offered by the actor's ISP.

Main Flow:

1. The assistant saves up and accumulates the elapsed time (the difference between end timestamp and the start timestamp of the user's being online).
2. The assistant pops up an alert message when the accumulated elapsed time is about to reach the threshold that is set in the pre-condition.

Additional Information: This assistant use case assumes that the user's online time involves only the user's web browsing activities, i.e. not including telnet and ftp.

4.3.4.9 RFA Mailto Assistant

Actor: End user

Description:

- This assistant allows the actor to specify a request for an answer (RFA) to all mailing addresses found on web pages browsed by the actor. If the actor is unable to find an answer to a question after browsing all related sites, this assistant allows the user to send a (the same) question to all of these related sites that contains a valid e-mail address that is not the webmaster's one.

Pre-conditions:

- This assistant needs to track all the contents of web pages.

Main Flow:

1. The assistant provides an interface that allows the actor to specify the RFA to be sent out.
2. The assistant informs the actor whether the action is successful or not.

Additional Information: None

4.3.4.10 Image Downloader

Actor: End user

Description:

This assistant automatically schedules itself to go to sites that the actor has browsed for the day and downloads images onto the actor's desktop. This assistant can save the actor time if the image downloading activity is done regularly.

Pre-conditions:

- The assistant needs to know the sites visited by the user for the day. Alternatively, this assistant can allow the actor to specify specific URLs.

Main Flow:

1. The assistant provides an interface that allows the actor to specify where to download the images from, whether a certain URLs or all sites browsed for a certain days.
2. The assistant schedules it-self to go to the designated URLs and download images.
3. The assistant notifies the actor when step 2 finishes.

Additional Information:

- The requirement of this assistant is derived from an actual question posted by Damon damonw@my-dejanews.com on April 14, 1999 on the *comp.lang.java.programmer* newsgroup.

> "I find myself going to the same web sites everyday to download images for work and I would like to fully automate the task, and possibly use a timer so the images can be waiting for me when I get to work.
>
> Can Java read the URL locations of numerous GID/JPG images, retrieve them, and write them to one directory, or does its security restrictions cause problems?"

4.3.5 Sandwich Use Cases

After going through the use cases of the application families, we now discuss the use cases of the framework itself. Each of the use cases below is described in detail using a similar but simplified template as above.

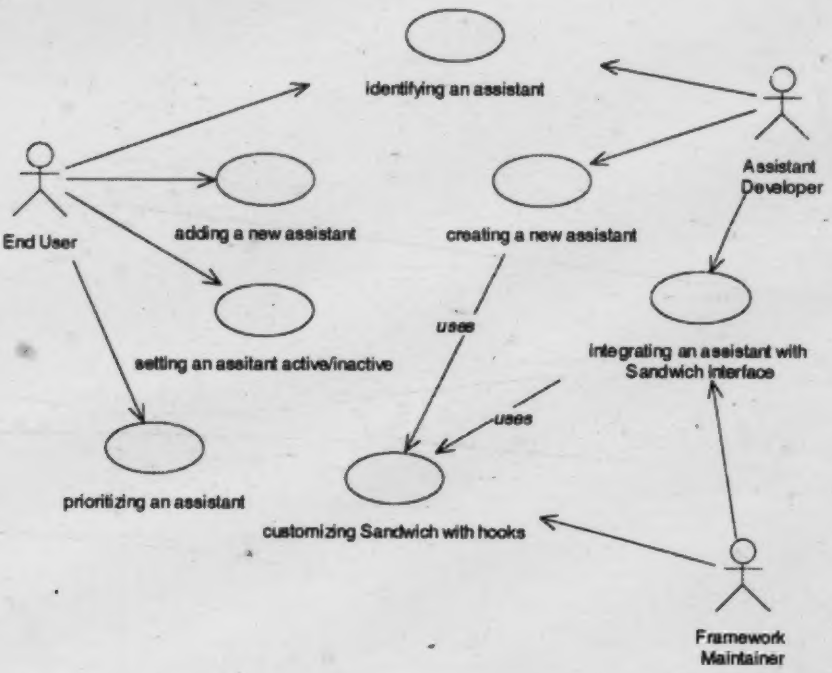

identifying an assistant
End User
adding a new assistant
creating a new assistant
Assistant Developer
setting an assitant active/inactive
uses
integrating an assistant with Sandwich interface
uses
prioritizing an assistant
customizing Sandwich with hooks
Framework Maintainer

Figure 10: ***Sandwich*** Use Cases

4.3.5.1 Identifying an Assistant

Actor: End user, Assistant Developer

Pre-conditions:

None

Main Flow:

1. The end user describes the desired assistantship to the assistant developer.
2. The assistant developer confirms the requirements by writing down the use case of this desired assistant using the template used above.
3. The end user accepts or refines the use case.
4. The assistant developer decides whether the new desired assistant is a candidate for ***Sandwich***'s pool of assistants.

Post-conditions:

- A conclusion on whether a new kind of assistantship can be created by either using an existing one or developing a new assistant for ***Sandwich***.

4.3.5.2 Creating a New Assistant

Actor: Assistant Developer

Pre-conditions:

- The assistant is currently not in ***Sandwich***'s assistant pool.
- The actor applied the use case Identifying an Assistant and the conclusion is true in that the assistant is a candidate of ***Sandwich***.

Main Flow:

1. The actor defines the functionality of this new assistant.
2. The actor creates the new assistant with the use case Customizing Sandwich with Hooks.
3. The actor documents the parameters to be configured for adding this new assistant to ***Sandwich***.

Post-conditions:

- The new assistant can be added to ***Sandwich***'s pool of assistants using the parameters documented.

4.3.5.3 Adding a New Assistant

Actor: End user

Pre-conditions:

The actor applied the use case Creating a New Assistant.

Main Flow:

1. The actor edits the property file to include this new assistant, using parameters documented down in the use case Creating a New Assistant.
2. The actor applies the use case Setting an Assistant to be Active/Inactive.
3. The actor restarts ***Sandwich*** and the new assistant will be initialized.

Post-conditions:

- ***Sandwich*** initializes this new assistant during start up.

4.3.5.4 Setting an Assistant to be Active/Inactive

Actor: End user

Pre-conditions:

- The assistant is currently registered with ***Sandwich.***

Main Flow:

1. According to the need, the actor changes the flag that indicates whether an assistant is active or not in a property file.
2. The actor restarts ***Sandwich*** to make the changes effective.

Post-conditions:

- The assistant becomes active or inactive depending on the new flag set.

4.3.5.5 Prioritizing An Assistant

Actor: End user

Pre-conditions:

- The assistant is currently registered with ***Sandwich***.
- The assistant acts on behalf of the user, i.e. delegating the user, for "postable" pages.

Main Flow:

1. According to the requirement, the actor changes the priority value for the assistant in a property file.
2. The actor restarts ***Sandwich*** to make the changes effective.

Post-conditions:

- Priorities of all active delegating assistants are unique.

4.3.5.6 Integrating an Assistant with Sandwich Interface (Admin Application)

Actor: Assistant Developer or Framework Maintainer

Pre-conditions:

- The assistant is added to ***Sandwich*** or is being created.

Main Flow:

1. The actor decides that the assistant has a GUI interface that can be integrated into the default ***Sandwich*** interface, the administrative application.
2. The actor enacts hooks that allow this assistant to integrate with the default ***Sandwich*** Interface.
3. The actor restarts ***Sandwich*** for testing and for changes to be effective.

Post-conditions:

- End user may now view the result of the assistant through this default ***Sandwich*** interface, the administrative application.

4.3.5.7 Customizing Sandwich with Hooks

Actor: Framework Maintainer

Pre-conditions:

- Hot spots of ***Sandwich*** that can be customized are documented with hooks.

Main Flow:

1. The actor determines which hot spot of ***Sandwich*** needs to be customized.
2. The actor follows the hook in customizing the hot spot, i.e. adding/removing capabilities or enabling/replacing default implementation.
3. The actor restarts ***Sandwich*** to test the new customization.

Post-conditions:

- Some capabilities are added/removed or default implementation is enabled/replaced.

4.4. *Sandwich* Subsystem, Design, Hooks and Rationale

Section 4.4 and its subsections describe the framework subsystem, design and its design rationale as well as the hook documentation for using the framework. Each sub-section encapsulates a core functionality of ***Sandwich***. Appendix D contains a summary on the UML notation that is used in this thesis, especially this section. We begin this section with an overview of the subsystem architecture of ***Sandwich***, as shown in Figure 11.

The two main components of ***Sandwich*** are the HTTP proxy and the pool of assistants. The proxy is responsible for taking an end-user request and fulfilling it using the pool of active assistants. The HTTP proxy initially loaded all registered assistant into memory by reading in the *assistant.properties* file using the persistence framework. Note that an assistant can be registered and not be active in ***Sandwich***. The ***Sandwich*** interface, referred as the administrative application, is secondary and provides an integrated GUI for interacting with the proxy (such as notifying the HTTP proxy when user has indicate shut down on the Sandwich Interface) and viewing any result of the assistants. The persistence framework is used in ***Sandwich*** for reading and updating the persistent storage such as the following property files,

- *sandwich.properties* that contains all configurable property of ***Sandwich*** in general,
- *assistant.properties* that contains all assistants in the pool, and
- other assistant-specific property files that are used by the assistants

The content of the *sandwich.properties* and *assistant.properties* files is included in Appendix E.

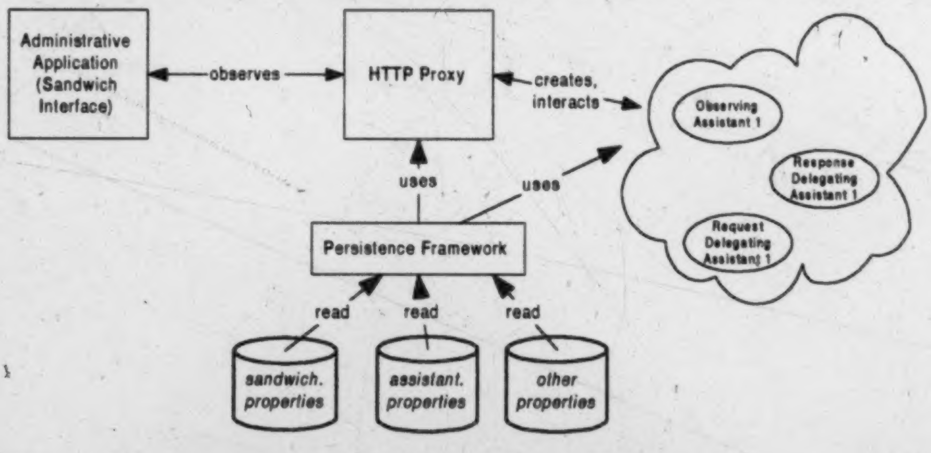
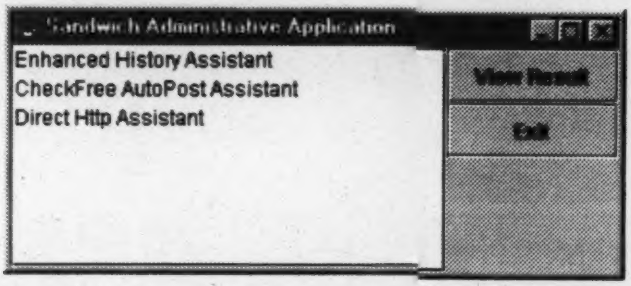

Administrative Application (Sandwich Interface)
observes
HTTP Proxy
creates, interacts
Observing Assistant 1
Response Delegating Assistant 1
Request Delegating Assistant 1
uses
uses
Persistence Framework
read
read
read
sandwich. properties
assistant. properties
other properties

Figure 11: *Sandwich* Subsystem Architecture

Thus, the ***Sandwich*** environment is essentially comprised of

- a set of classes that made up the HTTP proxy, ***Sandwich*** interface (Admin Application), persistent framework, registered and active assistants, and a set of property files, and
- the support to create new assistants

In order to deploy ***Sandwich***, the end user first configures his/her web browser such that all browsing activities go through ***Sandwich***'s HTTP proxy. For example, in Netscape this can be easily accomplished through the Edit/Preference/Advanced/Proxy screen. The end user then invokes the command to start ***Sandwich***, where the HTTP proxy starts as a daemon and the administrative application starts as a GUI application. The latter will bring up a small window that shows all registered assistants by their names, see Figure 12. The persistent framework is loaded into memory and used by both the proxy and the administrative application to read any property files. Registered assistants are instantiated; the HTTP proxy uses only assistants that are set to be active when fulfilling the user's HTTP requests.



Enhanced History Assistant
CheckFree AutoPost Assistant
Direct Http Assistant

Figure 12: *Sandwich* Interface, the Administrative Application

Another important facet of ***Sandwich*** is the support for creating new assistants. ***Sandwich*** contains a set of hooks in creating new personal web assistant. The initial step of this process focuses on identifying the assistant and confirming whether or not ***Sandwich*** can support it. During this step, ***Sandwich*** might be used (as is) to monitor the user's browsing activities and to provide an in-depth analysis on the data transmission that are happening for a particular site. When this assistant's requirement gathering and analysis phase is finished, a conclusion is made as to whether or not ***Sandwich*** can support this new assistant. If so, the next step is to create the new assistant using the documented hooks of ***Sandwich***. First, determine the type of this new assistant. The two types that are currently considered in ***Sandwich*** are observing and delegating assistants. If the new assistant simply monitors and provides a specific output based on the user's browsing activities, then the new assistant is an observing assistant. If the new assistant monitors and delegates on the user's behalf for "postable" pages (e.g. forms that have a "submit" button), then the new assistant is a delegating assistant. Based on the type of this new assistant, one or more documented hooks of ***Sandwich*** should be enacted to create this new assistant. After the development of the assistant is finished, registered the new assistant with ***Sandwich*** by adding new entries to the *assistant.properties* file. Finally, restart ***Sandwich*** such that this new assistant is now a candidate in the pool of assistants. Section 4.5 includes the details on the steps in creating new assistant.

Subsystem Keywords

From the subsystem architecture, we have chosen the following keywords to represent the main components of ***Sandwich***. These keywords will be used in the "Area" field of hook documentation.

- keyword **proxy** covers the HTTP proxy component
- keyword **personal assistant** covers the pool of assistant component
- keyword **administrative** covers the administrative application component
- keyword **utilities** covers the persistence framework as well as other support of ***Sandwich*** such as logging and the regular expression

UML Stereotypes

One of the features of UML is the ability to attach *stereotype* to classes. A *stereotype* is a type of model element defined in the model itself. The basic information content and form of a *stereotype* are the same as an existing base model element but with an extended meaning and different usage [RJB99]. We have included this brief section on *stereotype* because the documentation of ***Sandwich*** has used the following three stereotypes. Figure 13 illustrated the notation for these.

(1) The ***prototype*** stereotype is used through out this thesis to indicate classes that have been created for prototypes of this thesis,

(2) The ***hook*** stereotype has been used to indicate the hot spot of the framework where there is a hook documentation (see remarks below),

(3) The ***default*** stereotype has been used to indicate the default implementation of a certain hook of the framework

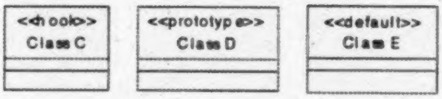


Figure 13: UML Stereotypes Deployed

Remarks:

Luyuan Liu's Master Thesis [Liu99] at SERL, which is currently in preparation, is investigate using UML *project* notation to model hook. For simplicity, we have chosen to use UML *stereotype* to diagrammatically show where the hooks are.

4.4.1 Sandwich *Startup*

When ***Sandwich*** first starts up, it creates and starts the proxy and the default administrative application. Both of them are registered as observers of each other. This approach allows them to communicate to each other via the observer pattern [GHJV95]. We describe the approach by first giving a dynamic view of how ***Sandwich*** starts, followed by a static view of the major classes involved and their relationships. We conclude this section with a discussion of the design rationale.

4.4.1.1 *Dynamic View*

The program begins execution in the *public static void main method* of the class *Sandwich* according to the following scenario. Please refer to Figure 14 and Figure 15 when going through the following.

1. Load the main properties file: *sandwich.properties* into a *SwProperties* object using the persistence framework.

2. Create the administrative application (a hot spot of ***Sandwich***). This is accomplished by invoking the *createAdminAppln* factory method of the class *AdminApplnFactory*, passing in the administrative class name obtained from the *SwProperties* object created in step 1. This administrative class name is a configurable parameter in the *sandwich.properties* file.

3. Using dynamic class loading and instantiation, *AdminApplnFactory* creates the appropriate administrative application or object.

4. Invoke the *starts()* method of the administrative application.

5. Create the *HttpProxy* (a frozen spot of ***Sandwich***), passing in the configurable port number and the assistant properties file name, both of which are obtained from the *SwProperties* object and thus are configurable parameters in the *sandwich.properties* file.

6. Get the *SwAdmin* model object from the administrative object created in step 3.

7. Register this *SwAdmin* object as an observer of the *HttpProxy* object created in step 5.

8. Register the *HttpProxy* object created in step 5 as an observer of the *SwAdmin* object.

9. Invoke the *starts ()* method of the *HttpProxy*.

Note: Step 1 will be elaborated in Section 4.4.6; steps 2, 3, 4 and 6 in the Section 4.4.5; steps 5 and 9 in Section 4.4.2.

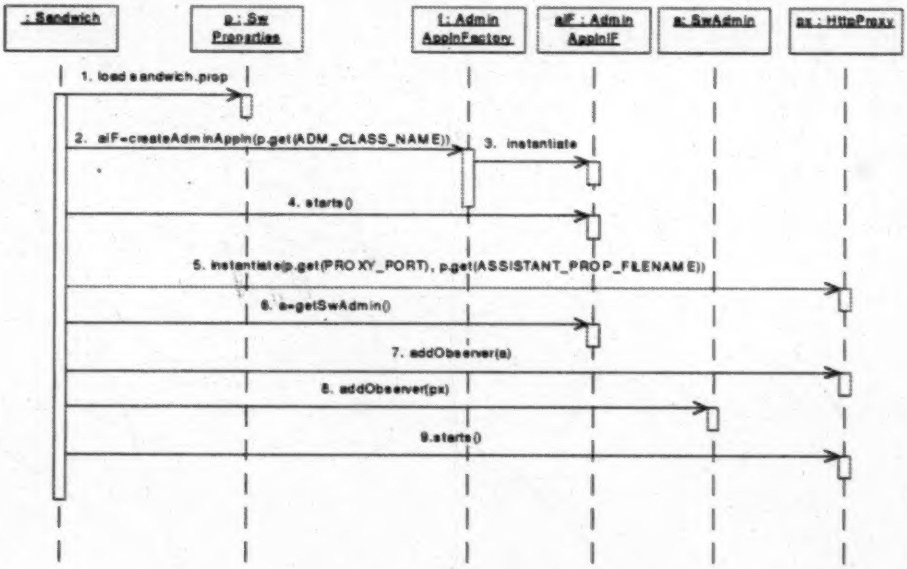

: Sandwich
p : Sw Properties
f: Admin AppInFactory
aIF : Admin AppInIF
a: SwAdmin
px : HttpProxy
1. load sandwich.prop
2. aIF=createAdminAppIn(p.get(ADM_CLASS_NAME))
3. instantiate
4. starts()
5. instantiate(p.get(PROXY_PORT), p.get(ASSISTANT_PROP_FILENAME))
6. a=getSwAdmin()
7. addObserver(a)
8. addObserver(px)
9.starts()

Figure 14: *Sandwich* Startup Dynamic View

4.4.1.2 Static View

Figure 15 shows a static view of the major classes that were involved during start up. *Sandwich* is the class that contains the *static public void main* method where the program begins execution. The *HttpProxy* and *SwAdmin* classes implement the JDK *Observer* interface and extend the Java Developer's Kit (JDK) *Observer* class in order for them to be observer as well as observable objects. Classes on the right hand side are part of the administrative component and will be further elaborated in Section 4.4.5; classes on the left-hand side are part of the proxy component and will be further elaborated in Section 4.4.2.

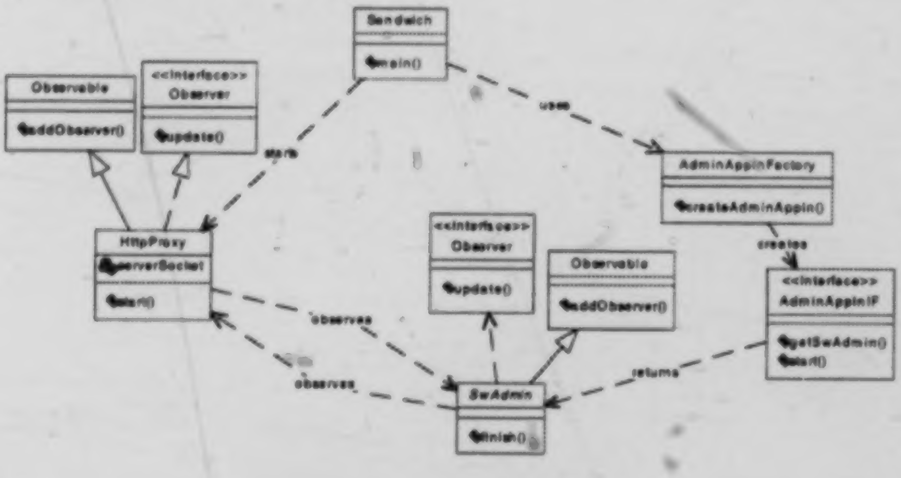

Sandwich
main()
Observable
addObserver()
<<interface>>
Observer
update()
starts
uses
AdminAppInFactory
createAdminAppIn()
HttpProxy
serverSocket
start()
<<interface>>
Observer
update()
Observable
addObserver()
creates
<<interface>>
AdminAppInIF
getSwAdmin()
start()
observes
observes
returns
SwAdmin
finish()

Figure 15: *Sandwich* Startup Static View

4.4.1.3 Design Rationale

Why does the class *Sandwich* exist?

Recall that the class *Sandwich* consists of the *static public void main()*, where the program execution begins. Initially, this main method existed in the class *HttpProxy* and there was no *Sandwich* class. This approach has implied, however, that ***Sandwich*** is the proxy itself. To reduce this misconception, the class *Sandwich* is introduced and in its main method a *HttpProxy* object is created and started. The administrative application is another object created and started by *Sandwich* in its main method. This delivers the clear message that ***Sandwich*** is comprised of the proxy and an administrative application. In addition, having adding this *Sandwich* class facilitates the future need to add new proxies support for other protocols such as FTP, NNTP and SMTP.

Why must the administrative application be registered as an observer of the proxy before the proxy is started?

Recall step 7 of the scenario above that the administrative application is registered as an observer of the proxy before the proxy is started in step 9. These steps must occur in this particular order because the administrative application needs to be notified in order that its interface can be updated when the proxy finishes loading all active assistants in its *start()* method.

4.4.2 HTTP Proxy

This component of ***Sandwich*** is responsible for setting up the pool of assistants. In this section, we will elaborate further on the dynamic and static view for two scenarios: one for starting up and one for fulfilling an HTTP request.

4.4.2.1 HttpProxy Startup

HttpProxy **Startup Dynamic View**

In conjunction with Figure 16 and Figure 17 the following section elaborates on the start up scenario of *HttpProxy* by ***Sandwich***.

1. The main program execution in class *Sandwich* creates (i.e. instantiates) the *HttpProxy* object. Recall that the port number and the assistants' configuration file name are passed in during this instantiation.

2. The main program execution then invokes the *start()* method of the *HttpProxy*.

3. With the given assistants' configuration file name, i.e. *assistant.properties*, *HttpProxy* loads the content of this file into a *SwProperties* object using the persistence framework.

4. Using this *SwProperties* object, all assistants (i.e. those with class names indicated in the *assistant.properties* file) will be instantiated using dynamic class loading.

5. For all active observing assistants, i.e. those that implement the *ObservingIF*, *HttpProxy* will registers them to be observers of the *HttpHeader* objects of interest. All the class names of interested HTTP headers by each observing assistant can be obtained from the above *SwProperties* object.

6. For all active delegating assistants, *HttpProxy* creates them and then puts them into priority queue.

7. Once loaded, *HttpProxy* fires the ASSISTANTS_UPDATE event so that observer such as the administrative application can refresh its interface to show all active assistants.

8. With the given port number, *HttpProxy* creates a JDK *ServerSocket* object.

9. *HttpProxy* invokes the *accept()* method of the *ServerSocket* object and this will continuously listen for incoming connections.

Note: Step 5 will be elaborated in Section 4.4.3; step 6 in Section 4.4.4.

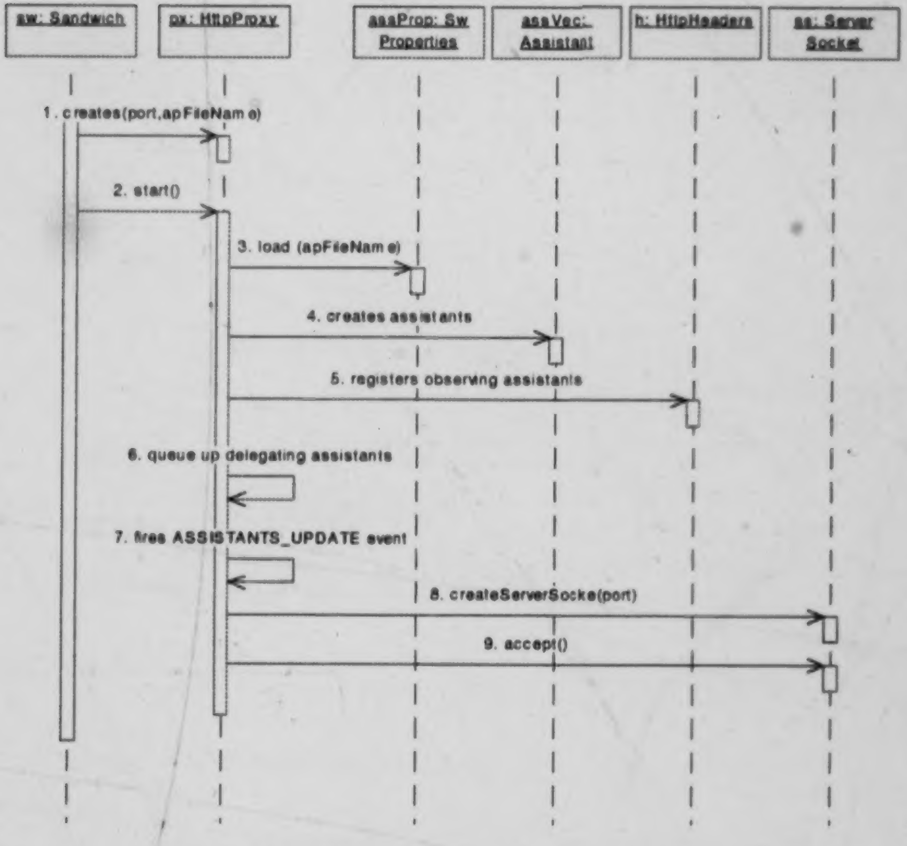

sw: Sandwich
px: HttpProxy
assProp: Sw Properties
assVec: Assistant
h: HttpHeaders
ss: Server Socket
1 . creates(port,apFileNam e)
2. start()
3. load (apFileNam e)
4. creates assist ants
5. registers observing assistants
6. queue up delegating assistants
7. fires ASSISTANTS_UPDATE event
8. createServerSocke(port)
9. accept()

Figure 16: HttpProxy Startup Dynamic View

HttpProxy Startup Static View

This section presents a static view of the above scenario. Each collaboration show in Figure 17 will be briefly described below along with the step number of the above scenario where the collaboration occurs.

- *HttpProxy and Assistant*

  HttpProxy is responsible to dynamically instantiate all registered assistants of ***Sandwich***, i.e. those whose class names appear in the *assistant.properties* file. (Step 4)

- *HttpProxy* and *ObservingIF*

  HttpProxy asks each assistant if it is an instance of *ObservingIF*. If so, then *HttpProxy* asks for its interested HTTP headers list and registers the assistant as an observer of each of these headers. (Step 5)

- *HttpProxy* and *HttpHeader*

  When registering an observing assistant as an observer of the header it is interested in, *HttpProxy* collaborates with each of the *HttpHeader* objects that are of interest to the observing assistant. (Step 5)

- *HttpProxy* and *DelegatingIF*

  HttpProxy asks each assistant if it is an instance of *DelegatingIF*. If so, then the assistant object is put into a queue, the order of which is based on the assistant's priority set. (Step 6)

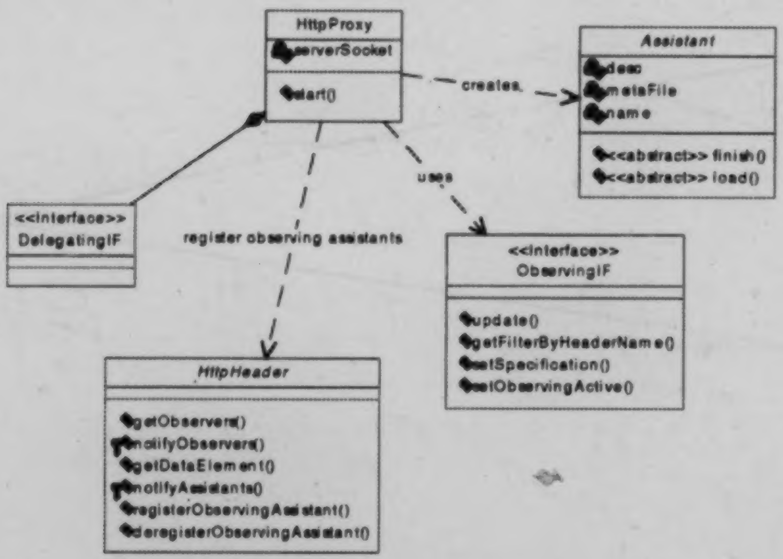

HttpProxy
serverSocket
start()
creates
Assistant
desc
metaFile
name
<<abstract>> finish()
<<abstract>> load()
uses
<<interface>>
DelegatingIF
register observing assistants
<<interface>>
ObservingIF
update()
getFilterByHeaderName()
setSpecification()
setObservingActive()
HttpHeader
getObservers()
notifyObservers()
getDataElement()
notifyAssistants()
registerObservingAssistant()
deregisterObservingAssistant()

Figure 17: HttpProxy Startup Static View

4.4.2.2 HttpProxy Servicing a Request

***HttpProxy* Servicing a Request Dynamic View**

In conjunction with Figure 18 and Figure 19, the following section describes how the *HttpProxy* fulfills an HTTP request.

1. When an incoming HTTP request is made by the end user, a new socket connection to the proxy is made. The proxy then instantiates a new *HttpServiceThread* object responsible for fulfilling this request.

2. The *HttpServiceThread* object constructs a *HttpRequest* object from the socket connection.

3. The *HttpServiceThread* object traverses through the queue of request delegating assistants.

4. When the object returned from step 3 traversal is of type *HttpResponse*, then the *HttpServiceThread* object proceeds to step 5.

5. The *HttpServiceThread* object proceeds to traverse the *HttpRequest* object through the queue of response delegating assistants.

6. The *HttpServiceThread* object fulfills the request by invoking the original *HttpRequest* object's *receiveResponse* method, passing in the final *HttpResponse* object from the step 4 or 5.

Note: Step 3, 4, and 5 will be elaborated in Section 4.4.4. The scenario for which observing assistants collaborates for each HTTP stream will be described in Section 4.4.3

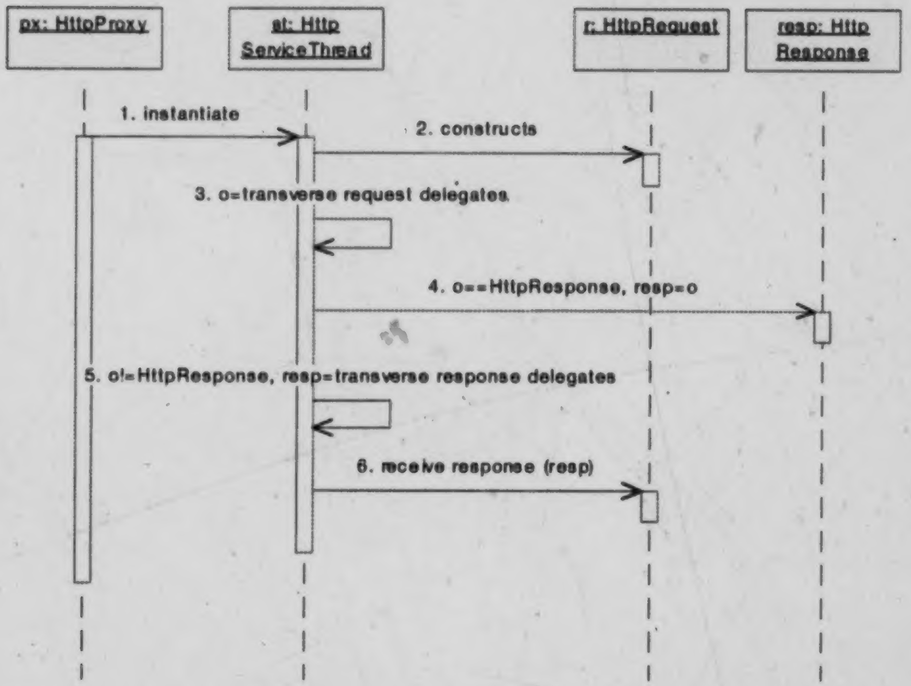

px: HttpProxy
st: Http ServiceThread
r: HttpRequest
resp: Http Response
1. instantiate
2. constructs
3. o=transverse request delegates
4. o==HttpResponse, resp=o
5. o!=HttpResponse, resp=transverse response delegates
6. receive response (resp)

Figure 18: HttpProxy Servicing Request Dynamic View

HttpProxy Servicing a Request Static View

This section includes a static view of the above scenario. Each collaboration shown in Figure 19 is briefly described along with the step number of the above scenario where the collaboration occurs.

- *HttpProxy* and *HttpServiceThread*

  For each incoming HTTP request, the *HttpProxy* creates a new thread, i.e. a *HttpServiceThread* object, to handle it. (Step 1)

- *HttpServiceThread* and *HttpRequest*

  Each *HttpServiceThread* object has a handle to the socket connection that contains the end user's HTTP request. The *HttpServiceThread* object reads the content of the socket and constructs a *HttpRequest* object from it (Step 2). Another collaboration occurs when the *HttpServiceThread* finishes the request by invoking the *receiveResponse* method of *HttpRequest* (Step 6).

- *HttpServiceThread* and *DelegatingIF*

  When the *HttpProxy* creates a *HttpServiceThread*, object references to the queues of delegating assistants, i.e. those that implemented the *DelegatingIF*, are passed in. The *HttpServiceThread* object traverses through these queues of delegating assistants with its constructed *HttpRequest* object. (Step 3, 4, and 5)

- *HttpServiceThread* and *HttpResponse*

  After the *HttpServiceThread* object traverses through the pool of delegating assistants, the final *HttpResponse* object is created and returned to the *HttpServiceThread*. (Step 5)

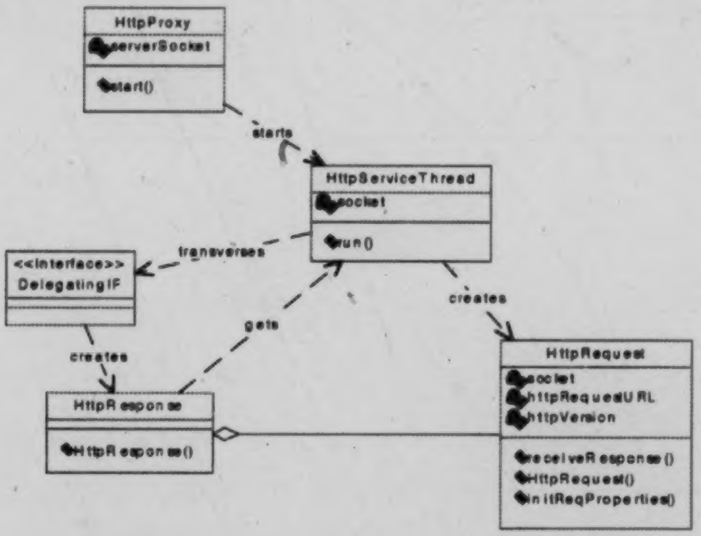

HttpProxy
serverSocket
start()
starts
HttpServiceThread
socket
run()
traverses
<<interface>>
DelegatingIF
creates
gets
creates
HttpRequest
socket
httpRequestURL
httpVersion
receiveResponse()
HttpRequest()
initReqProperties()
HttpResponse
HttpResponse()

Figure 19: HttpProxy Servicing Request Static View

4.4.3 Observing Assistant

Observing assistant functions as an observer by recording required data from a user's browsing activities and producing specific output from it. The three main data elements that exist in an HTTP request and response pair are (1) HTTP Headers, (2) HTTP Content (i.e. the resources itself – gif and html pages) and (3) HTTP META Tags that appear in the head portion of the HTML content itself. Both the content and META tags appear only in the HTTP response stream; HTTP Headers appear in both the request and response stream. Appendix A contains a list of all valid HTTP Headers, while Appendix B elaborates on HTTP META Tags.

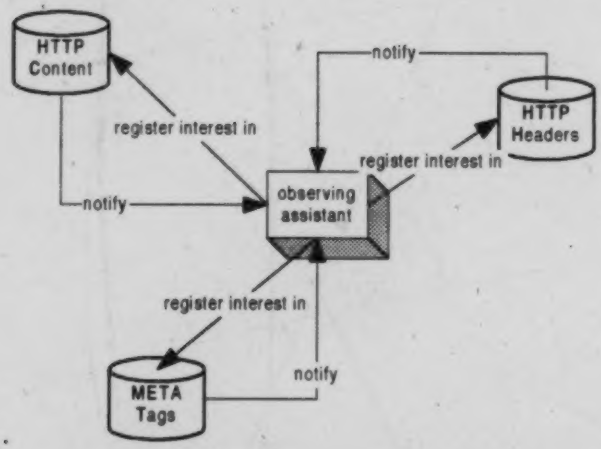


Figure 20: Data for Observing Assistants

In this section, we

(1) present a dynamic view of how *HttpProxy* creates and initializes observing assistants,

(2) present a dynamic view of how observing assistants get its event notification when one of its registered headers contain interesting content appears in a HTTP stream,

(3) provide a static view of major classes that are involved here and their collaborations,

(4) outline the **New Observing Assistant Hook**,

(5) describe an implementation example of the **New Observing Assistant Hook**, the Enhanced History Assistant prototype,

(6) discuss any relevant points of the design rationale

4.4.3.1 Observing Assistant Initialization

The following scenario, in conjunction with Figure 21, describes the dynamic view of how observing assistants are initialized by *HttpProxy* during startup. Thus, it elaborates on step 5 of Figure 16 where the proxy component of ***Sandwich*** registers observing assistants to be observers of their interested HTTP headers. Given an observing assistant, its initialization is successful when the *registerObservingAssistant* method of all the interested HTTP headers of the assistant are invoked, i.e. the final step 6. However, in order to accomplish this, several steps (from 2 to 5) are required to create JDK *Class* and *Method* objects using the JDK reflection API. Using this approach makes the coding very generic and can handle the registration of all observing assistants to their interested HTTP headers in only a few lines of code.

1. For each assistant created, the *HttpProxy* checks if the assistant is an observing assistant by asking if the assistant is an instance of *ObservingIF*. If so, then the following steps proceed. Otherwise, the *HttpProxy* proceeds with the next assistant.

2. The *HttpProxy* creates a JDK *Class* object for *ObservingIF* using the static *forName* method of the class *Class*.

3. The *HttpProxy* creates a JDK *Class* object for each of the interested HTTP headers of the assistant. The class names for these headers are obtained from the *Assistant* itself, through its *AssistantSpecification* containment.

4. For each class object of the assistant's interested HTTP headers created in step 3, get the JDK *Method* object with the method name as *registerObservingAssistant* and argument type as *ObservingIF*.

5. For each class object of the assistant's interested HTTP headers, create a new instance.

6. For each interested HTTP header object, invokes the *registerObservingAssistant* method dynamically, passing in the corresponding assistant object.

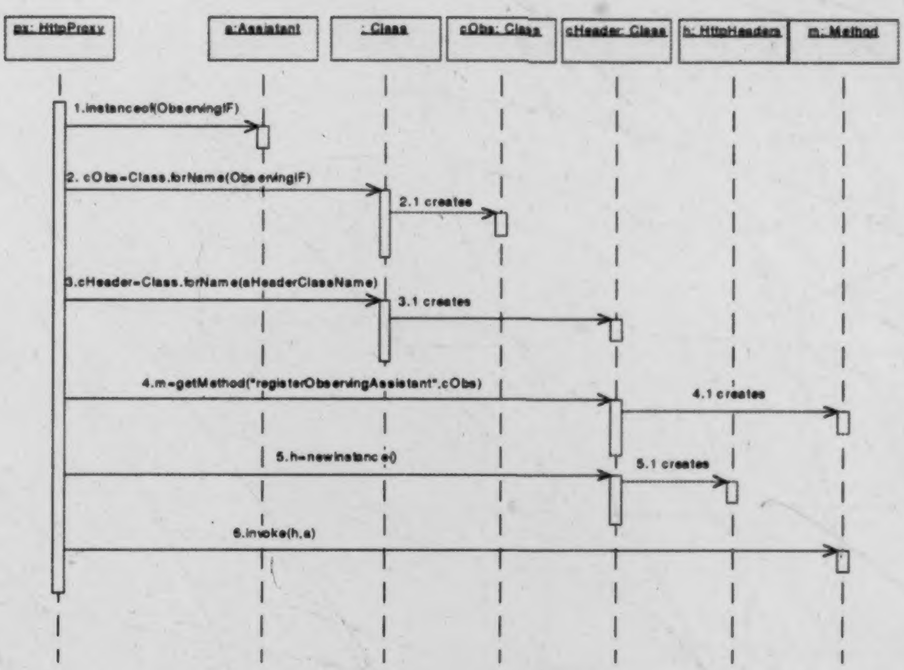

px: HttpProxy
a:Assistant
: Class
cObs: Class
cHeader: Class
h: HttpHeaders
m: Method
1.instanceof(ObservingIF)
2. cObs=Class.forName(ObservingIF)
2.1 creates
3.cHeader=Class.forName(aHeaderClassName)
3.1 creates
4.m=getMethod("registerObservingAssistant",cObs)
4.1 creates
5.h=newInstance()
5.1 creates
6.invoke(h,a)

Figure 21: Observing Assistant Initialization

4.4.3.2 Observing Assistant Notification

The following scenario, in conjunction with Figure 22, describes how an observing assistant is notified when one of its interested HTTP headers, the *HttpRequestURL*, exists in an HTTP request stream. The same scenario is applicable to all other supported headers as well as the HTTP response stream. Both *HttpRequest* and *HttpResponse* objects contain a set of *HttpHeader* objects.

1. Recall that the servicing thread for each incoming HTTP request stream creates and constructs a *HttpRequest* object from the socket connection.

2. During the construction of the *HTTPRequest* object, all headers (that appear in the request stream) will have a corresponding object created.

3. A more specific example of step 2 is that a *HTTPRequestURL* object is created and set to have the URL that the user has requested, e.g. http://www.cs.ualberta.ca/. Note that

HttpRequestURL object is also considered to be of type *HTTPHeader* because *HTTPRequestURL* is a subclass of *HTTPHeader*.

4. The *HTTPRequestURL* object then go through its list of observing assistant, checking if its current data element, i.e. http://www.cs.ualberta.ca in this example, matches the filter expression of the assistant for the class name *HTTPRequestURL*. This latter is accomplished through the regular expression API described in Section 4.4.8.

5. If so, then the assistant's *update()* method is invoked. It is now up to the assistant to handle this notification inside its *update()* implementation. The data available to the assistant in this *update()* method includes the thread number of the *HttpServiceThread* responsible for this request and the *HttpHeader* object.

So, generally speaking, each *HttpHeader* object will go through its list of observing assistants asking each assistant if the header's current data element is interested by the assistant (i.e. matches the assistant's regular expression). If so, then the assistant's *update()* method is invoked. Thus, ***Sandwich*** pre-defines the flow of control for notifying observing assistants and the actual implementation in handling these notifications is deferred to the assistant developer, see the template pattern by [GHJV95].

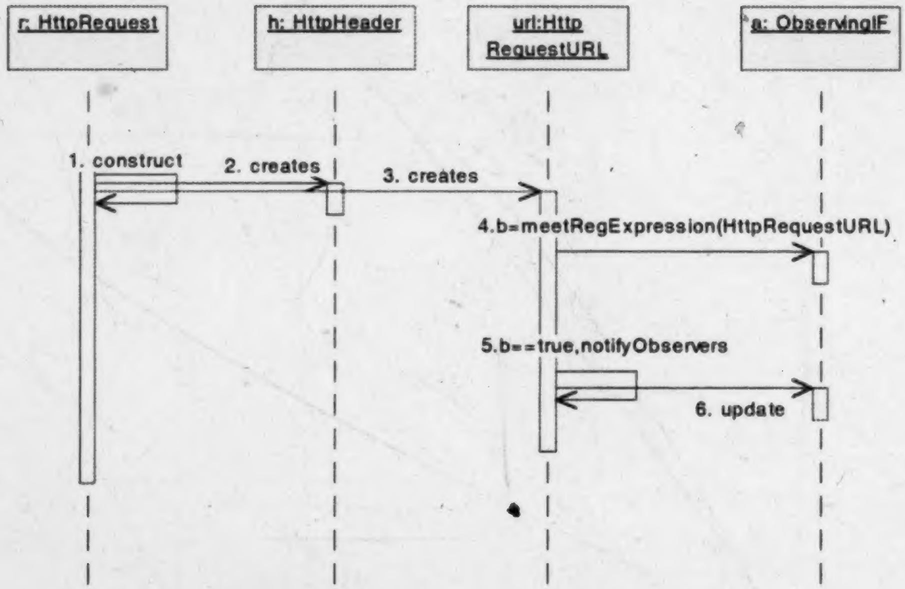

r: HttpRequest
h: HttpHeader
url:Http RequestURL
a: ObservingIF
1. construct
2. creates
3. creates
4.b=meetRegExpression(HttpRequestURL)
5.b==true,notifyObservers
6. update

Figure 22: Observing Assistant Notification Dynamic View

4.4.3.3 Observing Assistant Static View

Figure 23 illustrates a static view of the major and stereotyped classes that are involved in supporting observing assistants.

Stereotyped classes here can be broken down into the following two groups:

- Class *NewObservingAssistant* shows where the hooks are for the **New Observing Assistant Hook.**
- Class *EnhancedHistoryAssistant* is a class created for the prototype of an observing assistant.

Major classes and their collaboration are included in the following:

- *HttpProxy* and *HttpHeader*

  See Section 4.4.2.

- *HttpProxy* and *Assistant*

  Recall that *HttpProxy* creates all assistant objects from the *assistant.properties* file during startup.

- *HttpHeader* and *ObservingIF*

  Each *HttpHeader* contains a data element that represents the value of the header, e.g. http://www.cs.ualberta.ca for the requested URL header. When the data element changes state, the *HttpHeader* will traverse through its list of registered observing assistant (i.e. those that implements the *ObservingIF*). The observing assistant will be notified when the new state of the data element is of interested to the assistant, as determined by the regular expression set for the header.

- *HttpRequest/HttpResponse* and *HttpHeader*

  Both *HttpRequest* and *HttpResponse* objects contain a set of HTTP headers that appear in the request or response stream, respectively.

- *HttpHeader* and *HttpRequestURL*

  An *HttpRequestURL* object represents the requested URL of the request stream. *HttpRequestURL* is a subclass of *HttpHeader* since the requested URL is simply a header (with value) that appears in the request stream. For other subclasses of *HttpHeader* that are supported, see Section 4.4.9.

- *ObservingIF* and *AssistantSpecification*

  Each observing assistant has an *AssistantSpecfication* object that gives the list of headers by their class names along with their regular expression set. Only headers that can potentially become interesting to the observing assistant based on its content are included in this list. In Java, an interface can only contain a final attribute; therefore, the containment of an *AssistantSpecification* is implemented in the concrete class that implements the *ObservingIF*.

- *AssistantSpecification* and *Hashtable*

  Class *Hashtable* is a utility class from JDK. An *AssistantSpecification* object is simply a data structure in the form of a hash table. The name column of this hash table contains a *String* type object for the header class name; the value column contains a *String* type object for the header's regular expression. Thus, an *AssistantSpecification* object is able to return the regular expression of a given header class name. The following figure is an example of an *AssistantSpecification* object for an assistant that is interested in (a) all URL ever requested by the user, (b) all requests with response codes equal to 200 for OK or 304 for Not Modified, or 404 for Not Found.

name	value
com.http.HttpRequestURL	.
com.http.HttpResponseCode	200|304|404

In the *assistant.properties* file, the following entry will be created for this assistant.

```
l_assistant_spec1=com.http.HttpRequestURL,.
l_assistant_spec2=com.http.HttpResponseCode,200|304|404
```

NOTE:

(1) The regular expression API assumed in this example is the default implementation that comes with ***Sandwich***. See Section 4.4.8.

(2) This example is actually from one of the prototype assistants, Enhanced History Assistant, described below.

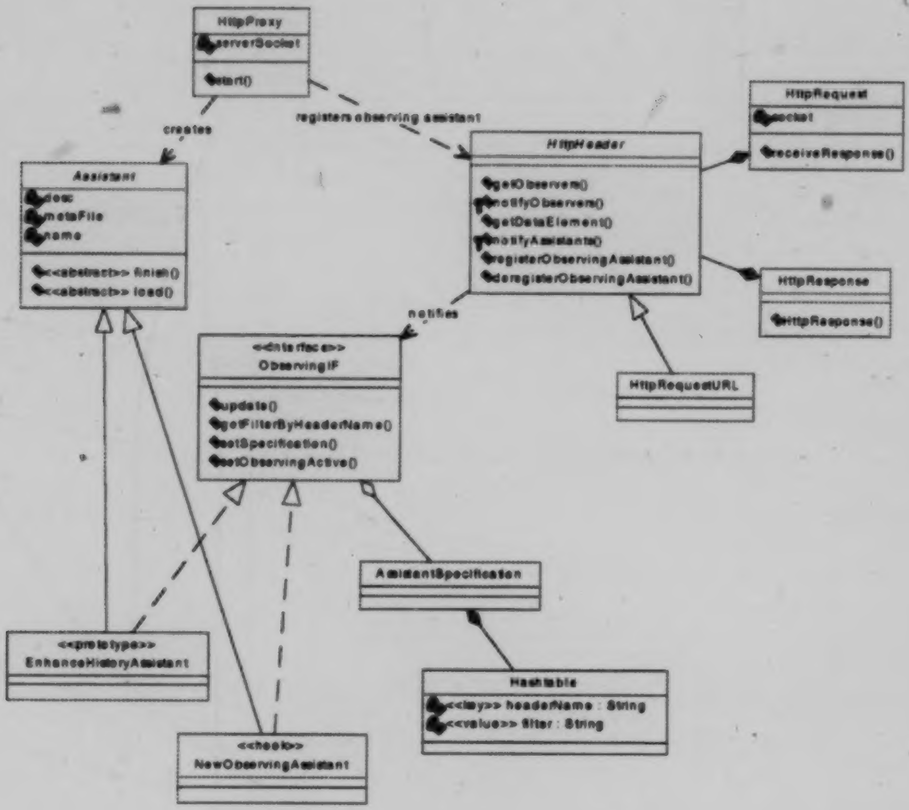

HttpProxy
serverSocket
start()
creates
registers observing assistant
Assistant
desc
metaFile
name
<<abstract>> finish()
<<abstract>> load()
HttpHeader
getObservers()
notifyObservers()
getDataElement()
notifyAssistants()
registerObservingAssistant()
deregisterObservingAssistant()
HttpRequest
socket
receiveResponse()
HttpResponse
HttpResponse()
notifies
<<interface>>
ObservingIF
update()
getFilterByHeaderName()
setSpecification()
setObservingActive()
HttpRequestURL
AssistantSpecification
<<prototype>>
EnhanceHistoryAssistant
Hashtable
<<key>> headerName : String
<<value>> filter : String
<<hook>>
NewObservingAssistant

Figure 23: Observing Assistants Static View

After describing the static and dynamic view on the behavior of observing assistants, we now look at the hook used to create a new observing assistant, **New Observing Assistant Hook.** We first outline this hook using the hook template proposed by [Froe96] and reviewed in Section 3.3. Then, we provide an example of how this hook is enacted by one of the prototype assistants, Enhanced History Assistant. Other assistants that are included in the use case and are considered as observing assistant include the Time Keeper, RFA Mailto Assistant and Image Downloader.

Name	New Observing Assistant Hook
Requirement	Add a new assistant that is interested in the user's browsing activities
Type	Adding Pattern
Area	Personal Assistants
Participants	Abstract class *Assistant*, interface *ObservingIF*
Uses	None
Pre-conditions	None

Changes	1. new subclass NewObservingAssistant of Assistant
	// to perform any assistant-specific initialization
	• NewObservingAssistant.load() overrides Assistant.load()
	// to release any system-resources held
	• NewObservingAssistant.finish() overrides Assistant.finish()
	2. Note: Identify which HTTP headers (by their class names) NewObservingAssistant is interested in
	3. Note: for each HTTP headers, identify its regular expression filter
	4. NewObservingAssistant implements ObservingIF interface
	• new property NewObservingAssistant.assSpec of type AssistantSpecification
	• NewObservingAssistant.setSpecification(AssistantSpecification) overrides ObservingIF.setSpecification(AssistantSpecification)
	writes NewObservingAssistant.assSpec
	• new property NewObservingAssistant.isActive of type boolean
	• NewObservingAssistant.setActive(boolean) overrides ObservingIF.setActive()
	writes NewObservingAssistant.isActive
	// return the value object of the assistant specification object, given headerName as the key
	• NewObservingAssistant.getFilterByHeaderName(String) overrides ObservingIF.getFilterByHeaderName(String)
	// to save data of interest either persistently or in memory, this method is
	// invoked for each data change made to each of the HTTP Headers
	• NewObservingAssistant.update() overrides ObservingIF.update()

	5. Note: register this assistant with ***Sandwich*** by updating the *assistant.properties* file with the following parameters • increase the number of assistants field, • indicate the fully qualified class name for NewObservingAssistant, • indicate the alias name, description, properties file name of the NewObservingAssistant, if any • indicate whether the NewObservingAssistant is active or not • indicate the number of HTTP headers that the NewObservingAssistant is interested in • indicate the specification of NewObservingAssistant (i.e. identified HttpHeaders(s) with its regular filter expression)
Post-conditions	A new assistant is added to ***Sandwich***'s pool of assistants
Comments	• If the assistant would like to support GUI output under the default administrative application, then apply the **New Assistant Result Window Hook**. • Step 5 can be optionally done through the administrative application, if supported

Hook 1: New Observing Assistant Hook

4.4.3.4 Enhanced History Assistant

Enhanced History Assistant is an example as well as the product of enacting the **New Observing Assistant Hook**. Using this hook, the following steps are followed during the creation of this new assistant, Enhanced History Assistant. Note that the number matches those of the hook "changes" section. This assistant also enacted the **New Assistant Result Window Hook** that will be elaborated in Section 4.4.5.

1. A new class *EnhancedHistoryAssistant* subclass of *Assistant* is created.
    - defined *load()* method to read any property file(s) of this specific assistant, e.g. *EnhHistory.prop*
    - defined *finish()* to close all files opened

    Note that the NewObservingAssistant referred in the hook is the *EnhancedHistoryAssistant* class when the hook is enacted.

2. Recall that this Enhanced History Assistant is interested to know all sites the user has ever visited and the status of each visit such as whether the pages are fetched from the remote server, from the browser cache or is not found in the remote server. Thus, the HTTP headers, along with their regular expression, that this assistant is interested in are
    - *HttpRequestURL*,.
    - *HttpResponseCode*,200|304|404

3. Class *EnhancedHistoryAssistant* implements the interface *ObservingIF*, as shown in Figure 23.
    - A new *AssistantSpecification* instance variable is declared in class *EnhancedHistoryAssistant* and defines the *setSpecification(AssistantSpecifaction)* be the setter of this variable.
    - A new boolean instance variable is declared in class *EnhancedHistoryAssistant* and defined the *setObservingActive(boolean)* be the setter of this variable.
    - Define the *getFilterByHeaderName(String headerName)* to return the value object of the *AssistantSpecification* object, e.g. spec.getFilterByHeaderName(headerName) where spec is the variable name for the new *AssistantSpecification* instance variable.
    - Define *update()* to save all notification by the header into a raw data file EnhHistoryRD.txt. For example, the first line of the following is saved when notification by *HttpRequestURL* is received. The parameters include the thread number, the time the notification is received, and the requested URL. The second line is saved when notification by *HttpResponseCode* is received. The parameters include the thread number, the time the notification is received, and the response code 200 for successful transmission from the remote web server.

```
Thread-0,Sun Jan 31 22:31:25 MST 1999,URL:http://java.sun.com/docs/books/tutorial/index.html
Thread-0,Sun Jan 31 22:31:27 MST 1999,RESPOND_CODE:200
```

4. Update the *assistant.properties* file with the following parameters. Assuming this is the first assistant of ***Sandwich***, the number of assistant field is therefore 1 (prefix of all property fields).

```
# the fully qualified class name (i.e. including
# package name) for the enhanced history assistant
1_assistant_classname=com.assistant.pool.enhancedHistory.EnhancedHistoryAssistant
```

```
# the alias name given to this assistant
1_assistant_name=Enhanced History Assistant

# description of what service this assistant provides
1_assistant_description= This assistant (a.k.a enhanced history assistant) will monitor all sites
you have visited and produced a statistic report for your browsed history

# this assistant's property file
1_assistant_propFileName=EnhHistory.prop

# indicates whether this assistant is active or not, as
# an observing type
1_assistant_activeObserving=1

# the number of HttpHeader classes this assistant is
# interested in
1_assistant_numSpec=2

# This means that all requested URLs are interested
1_assistant_spec1=com.http.HttpRequestURL,.

# This means that all responses with response code in
# 200, 304 or 404 are interested
1_assistant_spec2=com.http.HttpResponseCode,200|304|404
```

4.4.3.5 Design Rationale

Why introduce a new interface *ObservingIF* instead of using the JDK *Observer* interface?

Using the JDK *Observer* interface will require all *HttpHeader* classes that are observable to extend JDK *Observable* class. This will imply that the observer list (i.e. observing assistants) of each *HttpHeader* is per instance based rather than per class based. The former will then require the registration of an observing assistant to every instance of *HttpHeader*. This is redundant in the context of ***Sandwich*** where all objects belonging to the same *HttpHeader* class share a common list of observers suffices.

4.4.4 Delegating Assistant

As the name implies, the role of this type of assistant is to delegate. There are two types of delegating assistant: request delegate and response delegate. **A request delegate** takes an *HttpRequest* object, acts on it, and outputs either (a) a modified *HttpRequest* object indicated as *HttpRequest'*, or (b) an *HttpResponse* object. **A response delegate** takes an *HttpResponse* object, acts on it and outputs a modified response object as indicated with *HttpResponse'*.

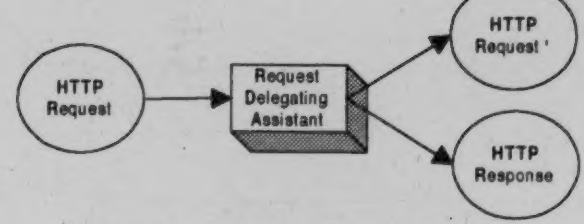


Figure 24: Request Delegating Assistant

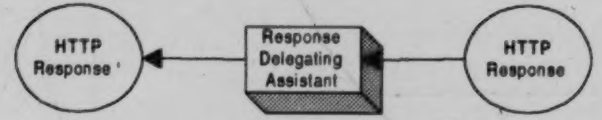


Figure 25: Response Delegating Assistant

In this section, we

(1) elaborate on the dynamic view of initializing delegating assistants,

(2) discuss the default delegating assistant of ***Sandwich***,

(3) describe how multiple response and request delegating assistants are handled by ***Sandwich***,

(4) present the static view of all major classes that are involved,

(5) outline the **New Request** and **Response Delegating Hook**,

(6) provides an example of how the CheckFree Auto Post Assistant enacted the **New Request Delegating Hook**,

(7) discuss any relevant points of the design rationale

4.4.4.1 Delegating Assistant Initialization

The following scenario, in conjunction with Figure 26, describes on the dynamic view of how request delegates are initialized during start up. Thus, it elaborates on step 6 of Figure 16 where the proxy component queues up delegating assistants after creating them, in the context of request delegates. The scenario below is also applicable to response delegates, the sequence diagram of which is shown in Figure 27.

1. For each assistant created, the *HttpProxy* checks if the assistant is a request delegating assistant by asking the assistant if it is an instance of interface *RequestDelegatingIF*.

2. Using the *SwProperties* object loaded for the *assistant.properties* file, the *HttpProxy* gets the request delegating active flag for this assistant. Recall that during the start up of the *HttpProxy* object, the *SwProperties* is created for the *assistant.properties* already.

3. The *HttpProxy* sets the active flag of the assistant to the value obtained from the *SwProperties* object.

4. The *HttpProxy* gets the priority, an integer value, of this assistant as a request delegate from the *SwProperties* object.

5. The *HttpProxy* sets the priority of this assistant to the value obtained in previous step.

6. Based on the assistant's priority, the *HttpProxy* inserts this assistant into the request delegate priority queue.

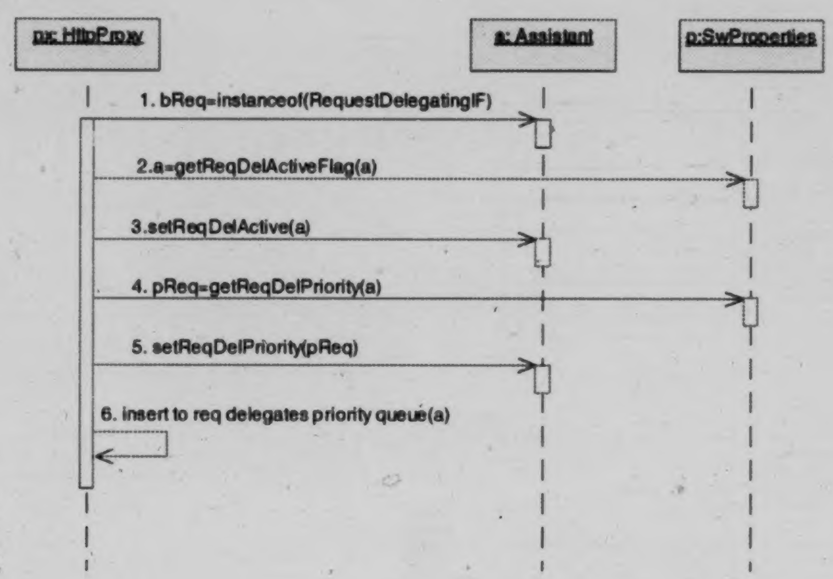

px: HttpProxy
a: Assistant
p:SwProperties
1. bReq=instanceof(RequestDelegatingIF)
2.a=getReqDelActiveFlag(a)
3.setReqDelActive(a)
4. pReq=getReqDelPriority(a)
5. setReqDelPriority(pReq)
6. insert to req delegates priority queue(a)

Figure 26: Request Delegate Initialization

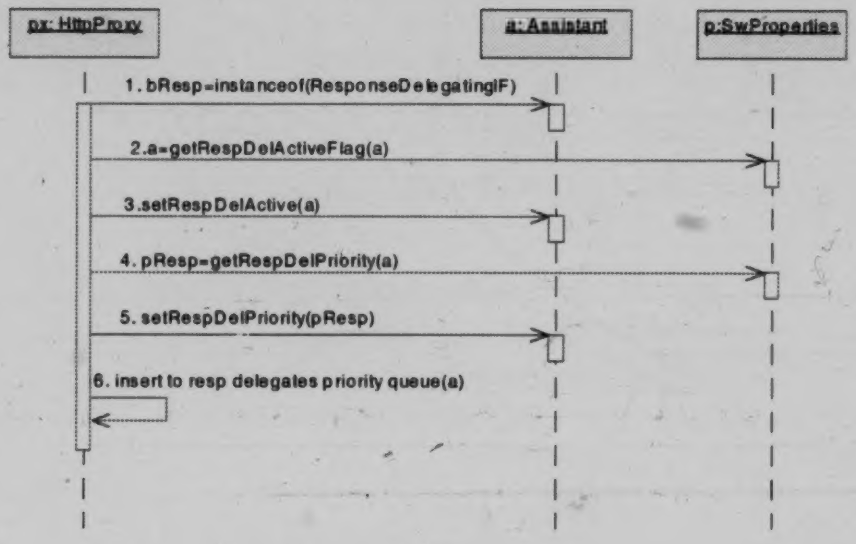

px: HttpProxy
a: Assistant
p:SwProperties
1. bResp=instanceof(ResponseDelegatingIF)
2.a=getRespDelActiveFlag(a)
3.setRespDelActive(a)
4. pResp=getRespDelPriority(a)
5. setRespDelPriority(pResp)
6. insert to resp delegates priority queue(a)

Figure 27: Response Delegate Initialization

4.4.4.2 Default Delegating Assistant

In this section, we will first give a high level view of where the default delegating assistant of ***Sandwich*** fits and then a low level (object level) dynamic view of how this default delegating assistant is active when no other request assistants are active within ***Sandwich***.

Default Delegating Assistant High Level View

Sandwich contains a default delegate assistant referred to as the *DirectHttpDelegate*. *DirectHttpDelegate* is an example of a request delegate. With no additional delegating assistants that are active, ***Sandwich*** is a very rudimentary HTTP proxy server operating with just *DirectHttpDelegate*. The framework constructs an *HttpRequest* object as directed from the user browser, and then sends this object to its delegating assistant pool that contains only the *DirectHttpDelegate* assistant. The role of this assistant is to go and fetch the requested resource (e.g. html and GIF file) from the remote web server and return the corresponding *HttpResponse* object. In the Figure 28 below, solid lines show that the framework is responsible for invoking the call while dotted lines indicate that the delegating assistant is responsible.

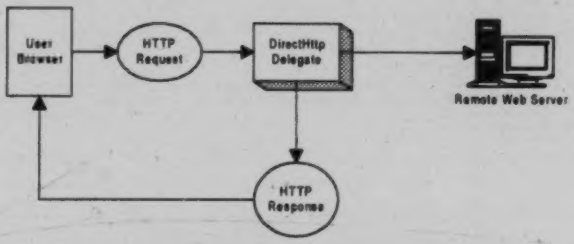


Figure 28: Default DirectHttpDelegate (High Level)

The following entries appear in the *assistant.properties* file by default. These entries trigger the creation of *DirectHttpDelegate* by *HttpProxy* during startup. Since *DirectHttpDelegate* is a type of request delegating assistant, i.e. it implements the *RequestDelegatingIF*, this default assistant will be put into the request delegate priority queue.

```
# this gives the fully qualified class name (i.e. including package name) that
# implements the default assistant
n_assistant_classname=com.assistant.DirectHttpAssistant

# this gives the name of this default assistant
n_assistant_name=Direct Http Assistant

# this indicates that the default assistant is active
n_ssistant_activeReqDelegating=1

# this gives the priority of this default assistant
n_assistant_ReqDelegatingPriority=1
```

`Note: "n" that begins each line is replaced by a number, determined by the number of assistants in the assistant.properties file. Lines that begin with "#" are comments only.`

Default Delegating Assistant Object Level View

The following scenario describes the dynamics for the situation in which ***Sandwich*** traverses the request delegates for each HTTP request, in the context that only the default delegating assistant is active. Thus, step 3, 4, and 5 of *HttpProxy* servicing a request scenario are elaborated.

1. Recall that the proxy instantiates a new *HttpServiceThread* for each incoming request, i.e. a socket connection.

2. The *HttpServiceThread* constructs a *HttpRequest* from the socket connection.

3. The *HttpServiceThread* gets the next assistant object from the request delegating priority queue. This queue returns the next request delegate with the highest priority. In this example, this queue returns the *DirectHttpAssistant* object because only the default assistant is active.

4. The *HttpServiceThread* asks the *DirectHttpAssistant* object whether a delegating opportunity exists. The implementation of the *reqDelOppExist()* method for class *DirectHttpAssistant* always returns true since ***Sandwich*** simply relays by default.

5. The *HttpServiceThread* asks the *DirectHttpAssistant* to delegate this request, i.e. invoking the *service()* method and passing in the *HttpRequest* object constructed.

6. *DirectHttpAssistant.service()* connects to the designated remote web server, gets the requested content from it, returns the response as a type of *Object*. During run time, the response object is a type of *HttpResponse*.

7. The *HttpServiceThread* checks if the returned object is a type of *HttpResponse*. This is true in this case.

8. The *HttpServiceThread* fulfills the request by invoking the *receiveResponse* method of the original *HttpRequest* object, passing it the returned *HttpResponse*.

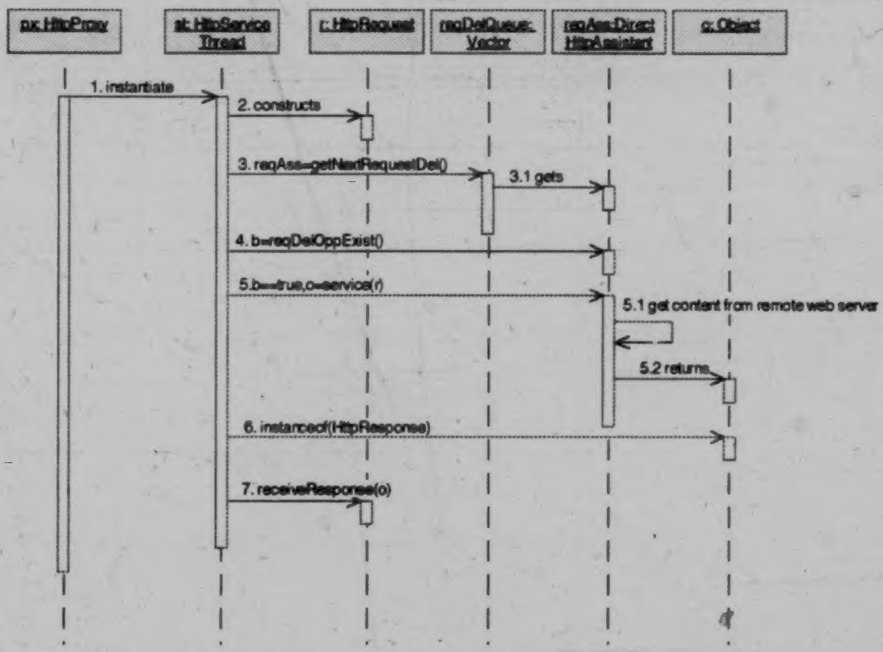

px:HttpProxy
st:HttpService Thread
r:HttpRequest
reqDelQueue: Vector
reqAss:Direct HttpAssistant
o:Object
1. instantiate
2. constructs
3. reqAss=getNextRequestDel()
3.1 gets
4. b=reqDelOppExist()
5.b==true,o=service(r)
5.1 get content from remote web server
5.2 returns
6. instanceof(HttpResponse)
7. receiveResponse(o)

Figure 29: Default *DirectHttpDelegate* (Object Level)

4.4.4.3 Multiple Request and Response Delegates

Now that we have seen a simple scenario of how ***Sandwich*** takes care of one delegating assistant, let us look at how multiple delegating assistants are handled by ***Sandwich*** conceptually. With multiple delegating assistants, ***Sandwich*** will sequentially pass the *HttpRequest* object to all active request delegates assistants until the output is an instance of *HttpResponse* type. Because of the default assistant, *DirectHttpDelegate*, all *HttpRequest* objects are guaranteed to be fulfilled with their corresponding *HttpResponse* objects.

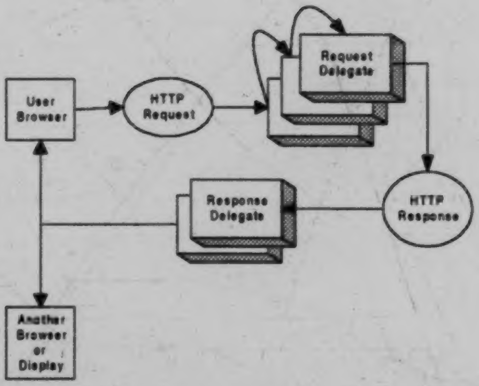


Figure 30: Multiple Delegating Assistants

4.4.4.4 Delegate Priority Queues

When multiple delegates are active, two queues are constructed by the proxy during start up, one for request delegates and one for response delegates. Figure 31 contains two examples where (a) is the request delegate priority queue, and (b) is the response delegate priority queue, both of which are constructed from the *assistant.properties* file with the following entries:

```
# this assistant is an active request delegate with
# priority of 2 and an active response delegate with
# priority of 1
1_assistant_classname=com.assistant.RequestDelegateA
1_asistant_activeReqDelegating=1
1_assistant_ReqDelegatingPriority=2
1_asistant_activeRespDelegating=1
1_assistant_RespDelegatingPriority=1

# this assistant is an active request delegate with
# priority of 1 and an active response delegate with a
# priority of 0
2_assistant_classname=com.assistant.RequestDelegateB
2_asistant_activeReqDelegating=1
2_assistant_ReqDelegatingPriority=1
2_asistant_activeRespDelegating=1
2_assistant_RespDelegatingPriority=0

# this default delegating assistant is an active
# request delegate but with the lowest priority, i.e. a
# priority of 0.
3_assistant_classname=com.assistant.DirectHttpAssistant
3_asistant_activeReqDelegating=1
3_assistant_ReqDelegatingPriority=0
```

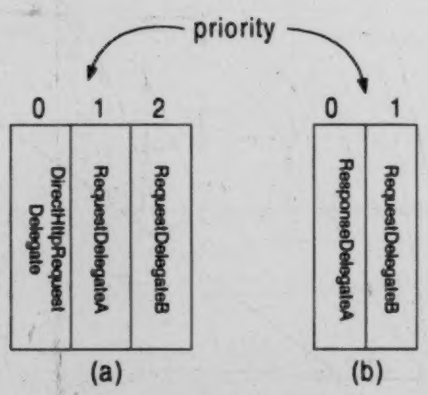

priority
0
1
2
0
1
DirectHttpRequest Delegate
RequestDelegateA
RequestDelegateB
ResponseDelegateA
RequestDelegateB
(a)
(b)

Figure 31: Delegate Priority Queues

An object level scenario of how multiple request delegates are handled in ***Sandwich*** is included in the following, in conjunction with Figure 32.

1. Recall that the proxy instantiates a new *HttpServiceThread* for each incoming request, i.e. a socket connection.

2. The *HttpServiceThread* constructs a *HttpRequest* from the socket connection.

3. The *HttpServiceThread* gets the next assistant with the highest priority from the request delegating priority queue.

4. The *HttpServiceThread* asks this delegate if an delegation opportunity exists by invoking the *public boolean reqDelOppExist(HttpRequest).*

5. (a) If step 4 returns false, then *HttpServiceThread* proceeds with the next request delegate in the queue by repeating step 3. (b) If step 4 returns true, then *HttpServiceThread* invokes the *service()* method of the delegate, where the request delegate executes its delegation.

6. The *HttpServiceThread* asks the return object from step 5(b) whether it is an instance of the *HttpResponse* class.

7. (a) If step 6 is true, then the *HttpServiceThread* continues with the scenario on which response delegates are traversed, and (b) If step 6 is false, the *HttpServiceThread* sets the return object as the current *HttpRequest* and continues with the next request delegate, i.e. repeats step 3.

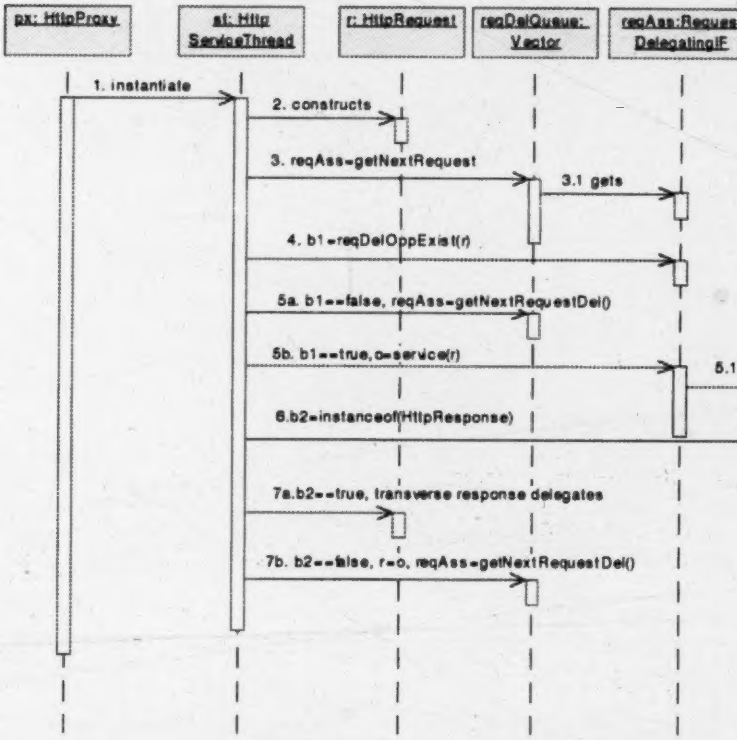

px: HttpProxy
st: Http ServiceThread
r: HttpRequest
reqDelQueue: Vector
reqAss:Request DelegatingIF
o: Object
1. instantiate
2. constructs
3. reqAss=getNextRequest
3.1 gets
4. b1=reqDelOppExist(r)
5a. b1==false, reqAss=getNextRequestDel()
5b. b1==true,o=service(r)
5.1 creates
6.b2=instanceof(HttpResponse)
7a.b2==true, transverse response delegates
7b. b2==false, r=o, reqAss=getNextRequestDel()

Figure 32: Multiple Request Delegates

Similarly, once a response object is returned in the request delegates traversal, ***Sandwich*** uses this response object as the input to the list of response delegates and traverses it. Having both of these chaining capabilities, ***Sandwich*** allows more than one delegating assistant to be registered. The following scenario, in conjunction with Figure 33, describes how multiple response delegates are handled in ***Sandwich***.

1. The *HttpServiceThread* gets the final *HttpResponse* object for a given *HttpRequest* from the above scenario, i.e. step 8 of Figure 32. This *HttpResponse* is set be the "current" *HttpResponse* object.

2. The *HttpServiceThread* gets the next assistant with the highest priority from the response delegating priority queue.

3. The *HttpServiceThread* asks this delegate if a delegating opportunity exist by invoking its *public boolean respDelOppExist(HttpResponse)* method.

4. If step 3 returns false, then the *HttpServiceThread* proceeds with the next response delegate by repeating step 2. (b) If step 3 returns true, then *HttpServiceThread* invokes the *service()* method of the delegate, passing in the current *HttpResponse* object. This is where the response delegate executes its delegation, e.g. modifying the response content pages for the Filtering Assistant example.

5. The return response object in step 4(b) is set to be the current *HttpResponse* object.

6. The *HttpServiceThread* repeats the above step 2-5 for all response delegates that are in the queue.

7. The final *HttpResponse* object is returned to its *HttpRequest* object.

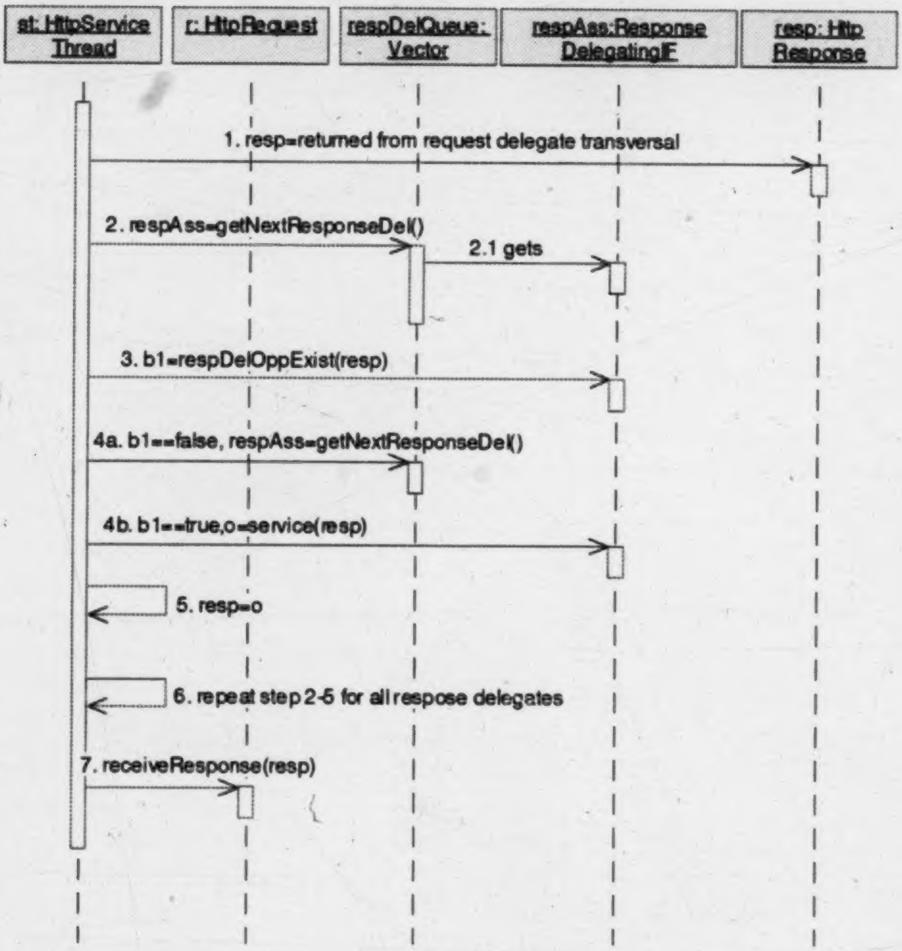

st: HttpService Thread
r: HttpRequest
respDelQueue: Vector
respAss:Response DelegatingIF
resp: Http Response
1. resp=returned from request delegate transversal
2. respAss=getNextResponseDel()
2.1 gets
3. b1=respDelOppExist(resp)
4a. b1==false, respAss=getNextResponseDel()
4b. b1==true,o=service(resp)
5. resp=o
6. repeat step 2-5 for all respose delegates
7. receiveResponse(resp)

Figure 33: Multiple Response Delegates

4.4.4.5 Delegating Assistant Static View

Figure 34 gives a static view of major and stereotyped classes that are involved in supporting delegating assistants described above. *Assistant*, *RequestDelegatingIF* and *ResponseDelegatingIF* are the only three major classes/interfaces that are required in supporting request and response delegates. ***Sandwich*** predefines the flow of execution for a typical request and response delegate.

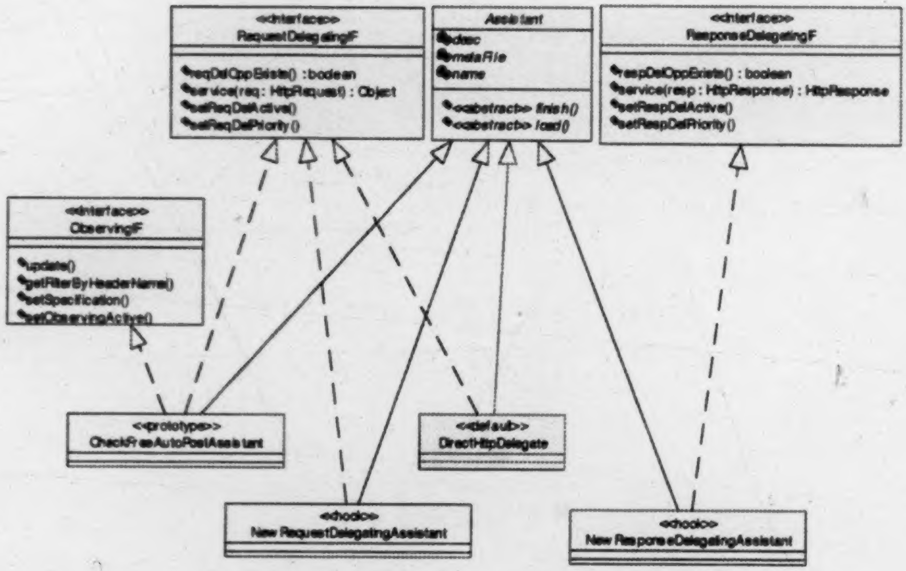

<<interface>>
RequestDelegatingIF
reqDelOppExists() :boolean
service(req : HttpRequest) : Object
setReqDelActive()
setReqDelPriority()
Assistant
desc
metaFile
name
<<abstract>> finish()
<<abstract>> load()
<<interface>>
ResponseDelegatingIF
respDelOppExists() : boolean
service(resp : HttpResponse) : HttpResponse
setRespDelActive()
setRespDelPriority()
<<interface>>
ObservingIF
update()
getFilterByHeaderName()
setSpecification()
setObservingActive()
<<prototype>>
CheckFreeAutoPostAssistant
<<default>>
DirectHttpDelegate
<<hook>>
New RequestDelegatingAssistant
<<hook>>
New ResponseDelegatingAssistant

Figure 34: Delegating Assistant Static View

The stereotyped classes here can be broken down into the following three groups:

- Class *NewRequestDelegatingAssistant* and *NewResponseDelegatingAssistant* show where the **New Request Delegating Assistant** and **New Response Delegating Assistant Hook** are, respectively.
- Class *DirectHttpDelegate* is a default implementation of the **New Request Delegating Assistant Hook** that is included in ***Sandwich***.
- Class *CheckFreeAutoPostAssistant* is a class created for the delegating prototype, i.e. one implementation of the **New Request Delegating Assistant Hook**. Note that this class also implements the *ObsevingIF*, i.e. enacted the **New Observing Assistant Hook**, because the assistant needs to monitor user activities first before acting on behalf of the user for postable pages of Check Free site.

Among the assistants described in the use case section, Auto POST Assistant, Search Summary Assistant, Simple Filtering Assistants, Change Monitor Assistant and Defer/Batch Post Assistant are examples of request delegates; PICS Filtering Assistant is an example of a response delegate. In this section, we begin with the **New Request Delegating Assistant Hook**, followed by an example of how this hook is enacted by the CheckFree Auto Post Assistant prototype. Finally, we outline the **New Response Delegating Assistant Hook** which is very similar to the **New Request Delegating Assistant Hook.**

Name	New Request Delegating Assistant Hook
Requirement	An assistant object that acts as a delegate for a user in certain circumstances, in particular, acting upon an upstream HTTP request
Type	Adding Pattern
Area	Personal Assistants
Participants	Interface *RequestDelegatingIF*, abstract class *Assistant* and *assistant.properties* file
Uses	None
Pre-conditions	None

Changes	1. new subclass NewReqDelAssistant of Assistant
	// to perform any assistant-specific initialization
	• NewReqDelAssistant overrides Assistant.load()
	// to release any system-resources held
	• NewReqDelAssistant overrides Assistant.finish()
	2. NewReqDelAssistant implements RequestDelegatingIF interface
	// to include the business logic in determining whether a delegation
	// opportunity exists; note that this is also where the assistant is
	// responsible to prompt for user confirmation, if any
	• NewReqDelAssistant.reqDelOppExists(HttpRequest) implements RequestDelegatingIF.reqDelOppExists(HttpRequest)
	// to perform the delegation; the framework invokes this callback if and
	// only if the reqDelOppExists(HttpRequest) returns true
	• NewReqDelAssistant.service(HttpRequest) implements RequestDelegatingIF.service(HttpRequest)
	3. Note: register this assistant with ***Sandwich*** by updating the *assistant.properties* file with the following parameters:
	• increase the number of assistant field,
	• indicate the fully qualified class name for NewReqDelAssistant,
	• indicate the alias name, description, property file name of

	NewReqDelAssistant, if any • indicate whether NewReqDelAssistant is active or not • determine the priority of NewReqDelAssistant
Post-conditions	A new request delegating assistant is added to ***Sandwich***'s pool of assistants
Comments	None

Hook 2: New Request Delegating Assistant Hook

4.1.4.6 CheckFree Auto Post Assistant

CheckFree Auto Post Assistant is an example and product of enacting the **New Request Delegating Assistant Hook**. Since this assistant also needs to monitor the user posting name value pairs for the Check Free site, this assistant is also an observing assistant and thus, also enacts the **New Observing Assistant Hook** as well. We first provide the steps involved in creating this new assistant as an observing assistant followed by the steps involved in creating this assistant as a request delegate.

As an observing assistant, the following steps matching the steps in the "changes" section of the **New Observing Assistant Hook** are followed:

1. A new class *CheckFreeAuotPostAssistant* subclass of Assistant is created.
    - defined *load()* to read the property file of this assistant, e.g. AutoPost.properties, and the raw data file, e.g. AutoPostRD.txt, that was saved by this assistant before, if any.
    - defined *finish()* to close any files opened.

   Note that the NewObservingAssistant used in the "changes" section of the **New Observing Assistant Hook** is the class *CheckFreeAutoPostAssistant* here.

2. This assistant is interested to know all name-value pairs posted by the user on the CheckFree site to get real time quotes. Thus, the HTTP headers that this assistant is interested in are the HTTP method, in particular the POST method, CheckFree URL(s), and the HTTP request body where all the name-value pairs are transmitted. Thus, the header classes and their regular expression filters are
    - *HttpMethod*,POST
    - *HttpRequestURL*,http://qs.secapl.com/cgi-bin/qs|http://qs-alt.secapl.com/cgi-bin/qs
    - *HTTPRequestBody*,.

3. Class *CheckFreeAutoPostAssistant* implements the interface *ObservingIF*, as shown in Figure 34.
    - Declared a new *AssistantSpecification* instance variable in class *CheckFreeAutoPostAssistant* and defined the *setSpecification* method as the setter of this variable.
    - Declared a new boolean instance variable in class *CheckFreeAutoPostAssistant* and defined the *setObservingActive* method as the setter of this variable.
    - Defined the *getFilterByHeaderName(String headerName)* to return the value object of the *AssistantSpecification* object, e.g. spec.getFilterByHeaderName(headerName) where spec is the variable name for the new *AssisstantSpecification* instance variable.
    - Defined *update()* to save all notification by the headers into a raw data file, *CheckFreeAutoPostRD.txt*. Some examples of this are included in the following. For each line, the thread number of the HTTP request, the date the notification is received, and the name-value pairs of the HTTP Post are saved. In this particular example, 3 pairs of name-value are used in CheckFree page where time, gif and tick are the names.

      ```
      •   Thread-9,Mon May 24 11:16:34 MDT 1999,
                {time=0000000927565940, gif=2, tick=t.atp t.bmo t.td t.nt t.nnc}
      •   Thread-0,Mon May 24 11:27:35 MDT 1999,
                {time=0000000927565940, gif=2, tick=t.v t.net.a t.oce.b t.lus.un t.brt}
      •   Thread-3,Mon May 24 11:28:35 MDT 1999,
                {time=0000000927566286, gif=3, tick=t.chp t.skl t.cos t.dmc t.bce}
      ```

4. Update the *assistant.properties* file, see Step 3 of the following steps when enacting the **New Request Delegating Assistant Hook.**

As a request-delegating assistant, whenever the user attempts to post to the Check Free URL, this assistant will pop up a dialog box, letting the user construct a new set of name-value pairs either with previously posted name-value pairs or changing them. This saves the user a great deal of time in any scenarios similar to the second scenario described in Chapter 1.

1. Same as step 1 of above when enacting the **New Observing Assistant Hook.**

2. Class *CheckFreeAutoPostAssistant* implements the *RequestDelegatingIF* interface.
    - define the *reqDelOppExists(HttpRequest)* method such that it will pop up a dialog box, letting the user to pick from a set of name-value pairs posted before

- define the *service(HttpRequest)* method such that the original *HttpRequest* object is modified to include the name-value pairs that user has selected above

3. Register this assistant with ***Sandwich*** by updating the *assistant.properties* file with the following parameters. Assuming the assistant is created as the Enhanced History Assistant, the number of assistant is then incremented to two. This will be used as the prefix to all property fields applicable to this new assistant.

```
# the fully qualified class name (i.e. including
# package name) for the CheckFree AutoPost Assistant
2_assistant_classname=com.assistant.pool.autoPost.AutoPostAssistant

# the alias name given to this assistant
2_assistant_name=Auto Post Assistant

# description of what service this assistant provides
2_assistant_description=This auto post assistant will prompt user whether an auto        post on
forms that have been previously filled out.

# this assistant's property file
2_assistant_metaDataFileName=AutoPost.properties

# indicates whether this assistant is active or not, as
# an observing type
2_assistant_activeObserving=1

# indicates whether this assistant is active or not, as
# a request delegating type
2_assistant_activeReqDelegating=1

# indicates the priority of this assistant, as a
# request delegating type
2_assistant_ReqDelegatingPriority=1

# the number of HttpHeader classes this assistant is
# interested in as an observing type
2_assistant_numSpec=3

# first header that this assistant is interested in as
# an observing type - all POST HTTP POST methods
2_assistant_spec1=com.http.HttpMethod,POST

# second header that this assistant is interested in as
# an observing type- only URLs of the CheckFree site
# that provide real-time quotes
2_assistant_spec2=com.http.HttpRequestURL,http://qs.secapl.com/cgi-bin/qs|http://qs-
alt.secapl.com/cgi-bin/qs

# third header that this assistant is interested in as
# an observing type: all request body that contains the
# name-value pairs
2_assistant_spec1=com.http.HttpRequestBody,.
```

Name	New Response Delegating Assistant Hook
Requirement	An assistant object that acts as a delegate for a user in certain circumstances, in particular upon an downstream HTTP response
Type	Adding Pattern
Area	Personal Assistants
Participants	Interface *ResponseDelegatingIF*, abstract class *Assistant* and *assistant.properties* file
Uses	None
Pre-conditions	None

Changes	1. new subclass NewRespDelAssistant of Assistant
	// to perform any assistant-specific initialization
	• NewRespAssistant.load() overrides Assistant.load()
	// to release any system-resources held
	• NewRespAssistant.finish() overrides Assistant.finish()
	2. NewReqDelAssistant implements ResponseDelegatingIF interface
	// to include the business logic in determining whether a delegation
	// opportunity exists; note that this is also where the assistant would
	// prompt for user confirmation, if any
	• NewReqDelAssistant.respDelOppExists(HttpResponse) overrides ResponseDelegatingIF.respDelOppExists(HttpResponse)
	// to perform the delegation; the framework invokes this callback if and
	// only if the respDelOppEx*ists(HttpResponse)* returns true
	• NewReqDelAssistant.service(HttpResponse) overrides ResponseDelegatingIF.service(HttpResponse)
	3. Note: register this assistant with ***Sandwich*** by updating the *assistant.properties* file with the following parameters:
	• increase the number of assistant field,
	• indicate the fully qualified class name for NewRespDelAssistant,
	• indicate the alias name, description, property file name of NewRespDelAssistant, if any

	- indicate whether NewRespDelAssistant is active or not - determine the priority of NewRespDelAssistant
Post-conditions	A new response delegating assistant is added to ***Sandwich***'s pool of assistants
Comments	None

Hook 3: New Response Delegating Assistant Hook

4.4.4.7 Design Rationale

Why do delegating assistants need to have a priority set?

Much of the rationale behind this priority flag lies in the way that the HTTP protocol works. In short, an HTTP stream contains a pair of request and response objects. To support multiple delegating assistants on the same stream, either the request or response stream, the framework must have some scheme of determining which delegate to invoke if more than one assistant is interested in delegating. There is various ways in doing this:

Alternative #1:
When a delegation opportunity by multiple delegates is detected, the framework can prompt the user with the list of delegates that are of interest for this delegation.

Alternative #2:
Using a blackboard approach [SG96] where all delegates bid for the delegation opportunity. This alternative is the same as a choosing the delegate on a first come first serve (FIFO) basis.

Alternative #3:
The user specifies the priority of a delegate when registering the assistant with the framework. This priority, which is simply an integer value, allows ***Sandwich*** to determine the order of delegating assistants to be invoked when a delegation opportunity exists. In ***Sandwich***, a delegate can be assigned with two different priorities when acting in both a request and response role.

We have chosen alternative #3 where the user decides ahead of time on which delegate has a higher priority over another. Alternative #1 might be overwhelming or burdensome to the user as he or she has to select a particular delegate whenever there is a delegation opportunity. Alternative #2 is too uncertain in terms of which delegate will get the bid or which will come first. If alternative #2 is implemented, the user must confirm with the approved delegate, like alternative #1. Alternative #3 was chosen over the other two mainly because the user is likely to want to have a particular delegate for a particular request. For example, it may be the case that a particular set of delegates are interested in the delegation opportunity based on a request for a particular URL and the highest priority should be set to the most preferred delegate.

4.4.5 Administrative Application (Sandwich Interface)

This component of ***Sandwich*** is simply an integrated GUI application that provides an interface to the proxy component as well as the pool of assistants of ***Sandwich***. At present, the two main services provided by the administrative application are

(1) informing ***Sandwich***'s proxy component when a particular event occurs, such as the shutdown event triggered when user clicks on the exit button,
(2) rendering of a selected assistant's panel, such as the output panel of an observing assistant

By default, ***Sandwich*** creates a Web Application Management Tool (WAMT) implemented in the class *com.admin.WAMT* as the default administrative application, the ***Sandwich*** Interface. Creation of this administrative component is a hot spot of ***Sandwich*** and thus framework users can replace this default administrative application with a new one. The **New Administrative Application Hook** describes the steps required in replacing this default WAMT with a new implementation.

In this section, we will

(1) present an overview of the major dynamic scenarios of this administrative component,
(2) provide a static view of major classes that shows the relationships of the major classes involved as well as on the stereotyped classes that are used for
    - hooking in a new administrative application,
    - the default administrative implementation,
(3) outline the **New Administrative Application Hook**,
(4) elaborate on the default administrative application, the ***Sandwich*** Interface

4.4.5.1 Dynamic View

The two main dynamic views of this administrative component will be described in this section. The first scenario describes how the communication occurs between this administrative component and the proxy component of ***Sandwich***, and the second scenario describes how the communication occurs between this administrative component and the pool of assistants of ***Sandwich***.

Communication with the Proxy

The communication between this administrative application and the proxy is through the observer pattern approach [GHJV95]. Basically,

a) As an object observable by the proxy, this default administrative application triggers events when it wants to broadcast to all its observers that the user has requested a particular event or its internal state has changed. For example, the following scenario shows WAMT firing the SHUT_DOWN_EVENT when the user indicates that he/she has selected to exit ***Sandwich*** through the interface. This then triggers a notification event to all registered observers of this default administrative application, including the *HttpProxy* component of ***Sandwich***. Recall in step 8 of the starts up scenario that the *HttpProxy* is registered as an observer of the administrative application. The *HttpProxy* object will then shut down cleanly, releasing all system resources.

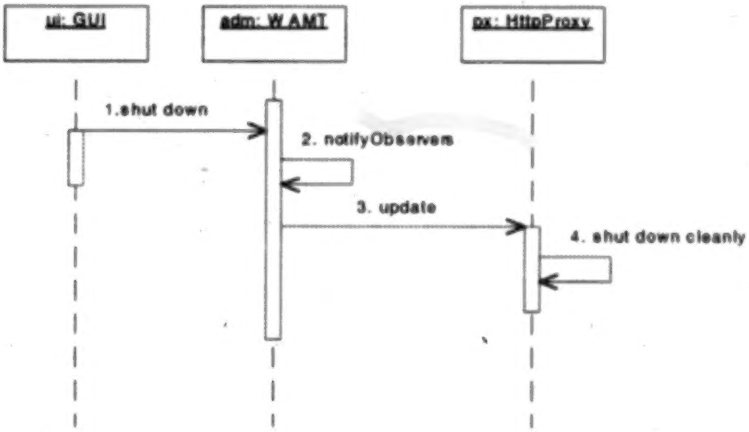


Figure 35: Administrative Application and HttpProxy

b) As an observing object of the proxy, this default administrative application is interested in the ASSISTANT_UPDATE_EVENT event that is triggered by the proxy when all assistants are being loaded.

Communication with the Pool of Assistants

Sandwich uses the template [GHJV95] pattern in achieving the communication between the administrative application and the pool of assistants. One of the services provided by the administrative application is to be able to view the result panel of assistants, if any. The following scenario in conjunction with Figure 36 elaborates on this.

1. User selects an assistant presented by the graphical interface of this administrative application, see Figure 12.
2. User clicks on the "View Result" button.
3. This triggers the *actionPerformed()* callback method of the button's listener.
4. This listener or handler of the "View Result" button will then ask the *WAMT* for the selected assistant,
5. This listener verifies if the selected assistant is an *AssistantAdminIF*, i.e. implements the *AssistantAdminIF* interface.
6. If step 5 is true, then the listener will ask the selected assistant for its result panel.
7. The listener renders the assistant's result panel in a window. If step 5 is false, a dummy message box saying that the assistant has not enacted the **New Assistant Result Window Hook** is popped up.

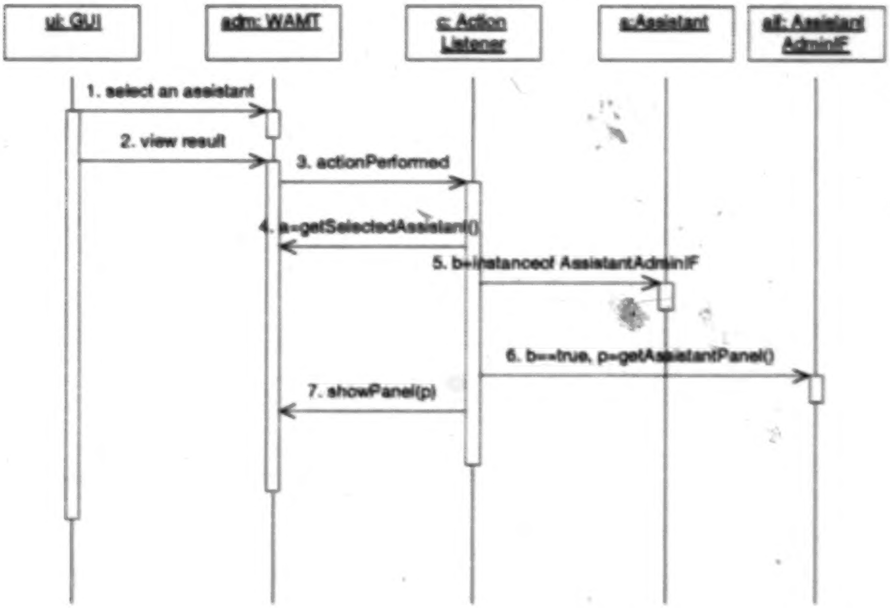

ui: GUI
adm: WAMT
c: Action Listener
a:Assistant
aI: Assistant AdminIF
1. select an assistant
2. view result
3. actionPerformed
4. a=getSelectedAssistant()
5. b=instanceof AssistantAdminIF
6. b==true, p=getAssistantPanel()
7. showPanel(p)

Figure 36: Administrative Application and Assistants

4.4.5.2 Static View

Figure 37 gives a static view of the major and stereotyped classes involved in the administrative component. Other classes that are shown in the figure (class *JFrame* and *Observable* and interface *Observer*) are from JDK and are included in the figure for the purposes of completeness only.

Stereotyped classes here can be broken down into the following two groups:

- Class *NewAdminAppln* and *NewSwAdmin* show where the hooks are for enacting the **New Administrative Application Hook**.
- Class *SwDefaultAdmin* and *WAMT* are default classes created for the default administrative application, which is one implementation of the **New Administrative Application Hook**.

Major classes and their collaboration are included in the following:

- Class *AdminApplnFactory*

This class dynamically creates an object that conforms to the *AdminApplnIF* interface based on the class name passed in as an argument of the class's main factory method. In Java, this is accomplished through the *Class.forName* and *newInstance()* methods of the JDK reflection capability. This class implemented the *abstract factory* creational pattern documented in [GHJV95]. The two recurring themes in applying this pattern are (a) they encapsulate knowledge about which concrete class the system uses, and (b) they hide how instances of these classes are created and put together. In short, they help to make a system independent of how its objects are created.

- Interface *AdminApplnIF*

  This interface declares all abstract methods that the framework expects from its administrative component.

- Class *SwAdmin*

  The class is the model of the administrative application expected by the framework. It is expected to be an observer as well as an observable.

Sandwich is solely aware of the abstract level classes or interfaces including only the *AdminApplnIF* interface and class *SwAdmin*. The latter is expected to implement the *Observer* interface and extends *Observable* such that it can be registered as an observer of the proxy component and be observed by the proxy, respectively.

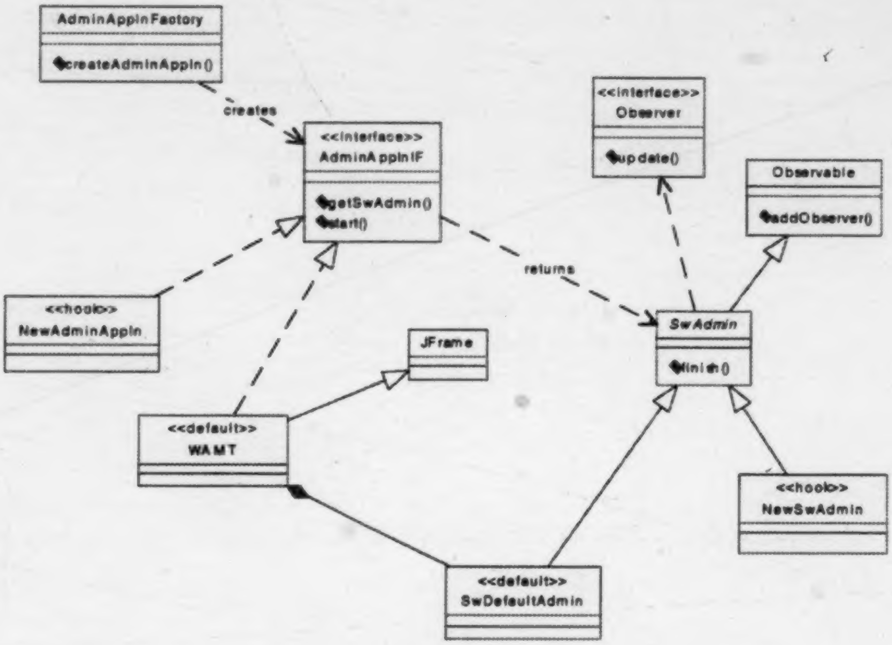

AdminApplnFactory
createAdminAppln()
creates
<<interface>>
AdminApplnIF
getSwAdmin()
start()
<<interface>>
Observer
update()
Observable
addObserver()
returns
<<hook>>
NewAdminAppln
JFrame
SwAdmin
finish()
<<default>>
WAMT
<<hook>>
NewSwAdmin
<<default>>
SwDefaultAdmin

Figure 37: Administrative Component Static View

Name	New Administrative Application Hook
Requirement	An administrative tool or interface for ***Sandwich***
Type	Replacing Pattern
Area	Administrative
Participants	Interface *AdminApplnIF*, abstract class *SwAdmin* and the *sandwich.properties* file
Uses	None
Preconditions	None
Changes	1. new class NewAdminApplication implements AdminApplnIF interface 2. new subclass NewSwAdmin of SwAdmin // define with handlers to all events expected from the proxy 3. NewSwAdmin.update() overrides SwAdmin.update() 4. NewAdminApplication.getSwAdmin() overrides AdminApplnIF.getSwAdmin() returns NewSwAdmin

	// to show the NewAdminApplication 5. NewAdminApplication.start() overrides AdminApplnIF.start() 6. Note: update a field in the *sandwich.properties* file: admin_application_ classname=NewAdminApplication's class name
Post-conditions	Default administrative application is replaced by NewAdminApplication
Comments	None

Hook 4: New Administrative Application Hook

4.4.5.3 Default Administrative Application

Figure 37 shows the default administrative application of ***Sandwich***, i.e. classes with the <<*default*>> stereotype. Basically, this default implementation is an example of enacting the **New Administrative Application Hook**. In summary, this is accomplished by the following:

1. Creates a new class *WAMT* that implements the *AdminApplnIF* interface. Note that
    - *WAMT* is the NewAdminApplication used in the **New Administrative Application Hook**, and
    - *WAMT* extends JDK *JFrame* and thus is a GUI application by definition.

2. Creates a new class *SwDefaultAdmin* that subclasses the *SwAdmin* abstract class. Note that *SwDefaultAdmin* is the NewSwAdmin used in the **New Administrative Application Hook**.

3. Define the *SwDefaultAdmin.update()* method to handle the anticipated event from the proxy such as the ASSISTANT_UPDATE event

4. Define the *WAMT.getSwAdmin()* method to return *SwDefaultAdmin*.

5. Define the *WAMT.start()* to render the GUI of *WAMT*

6. Update a field in the *sandwich.properties* file:
    ```
    admin_application_ classname=com.admin.default.WAMT
    ```

The following is a hook that is applicable only when ***Sandwich*** uses the default administrative application. It basically allows an assistant to hook in a GUI that can be invoked through the default ***Sandwich*** interface. This hook is enacted by one of the prototype assistants, Enhanced History Assistant.

Name	New Assistant Result Window Hook
Requirement	An assistant wants to display a result panel through ***Sandwich*** administrative component.
Type	Adding Pattern
Area	Administrative
Participants	Interface *AssistantAdminIF*, anAssistant *Assistant*
Uses	None
Preconditions	• the default administrative application WAMT is used • anAssistant is an existing *Assistant* object in ***Sandwich***

Changes	// anAssistant is currently an instance of Assistant class
	1. anAssistant implements AssistantAdminIF interface
	// return a *JPanel* that contains the assistant output panel, which gets
	// rendered when user selects the "View Result" button
	2. anAssistant.*getAssistantPanel()* overrides AssistantAdminIF.getAssistantPanel()

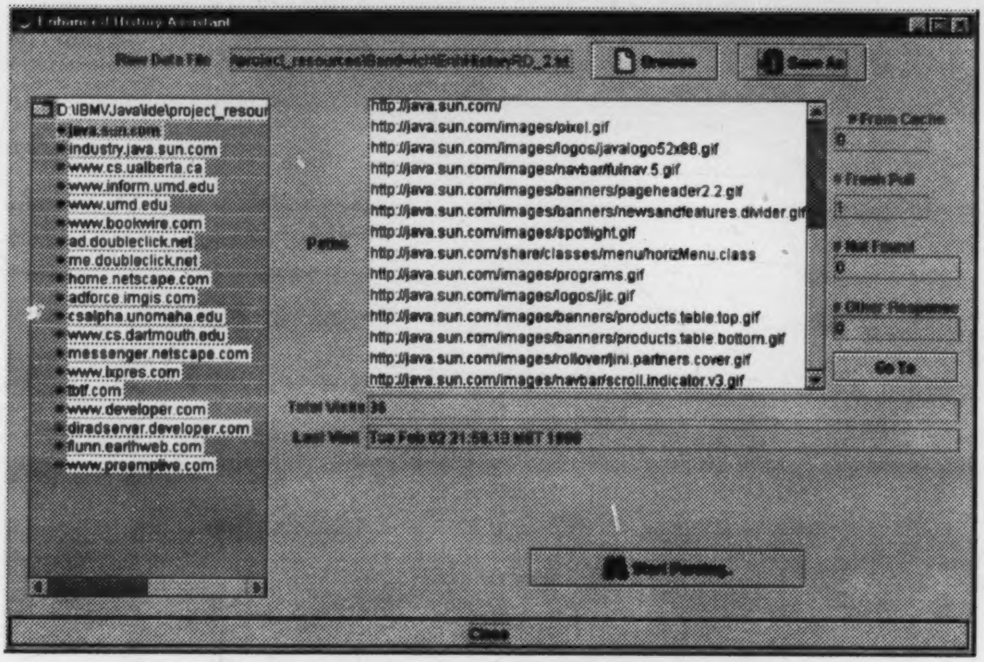
Post-conditions	When user selects anAssistant and click on the "View Result" button, then the panel returned by the *getAssistantPanel()* method will get rendered.
Comments	None

Hook 5: New Assistant Result Window Hook

The Enhanced History Assistant enacted this hook such that its result window can be accessed through the default ***Sandwich*** Interface, the administrative application. Basically,

1. The class *EnhancedHistoryAssistant* implements the interface *AssistantAdminIF*
2. Define the *getAssistantPanel()* method to return a JDK *JPanel* object that contains the output panel.

After the above creation is finished, restarts ***Sandwich***, select "Enhanced History Assistant" in the window of the default administrative application, as shown in Figure 12, and clicks on the "View Result" button. A window (see Figure 38) appears, which contains the *JPanel* of the Enhanced History Assistant.



Raw Data File: \project_resources\Sandwich\EnhHistoryRD_2.txt

D:\IBMVJava\ide\project_resour
- java.sun.com
- industry.java.sun.com
- www.cs.ualberta.ca
- www.inform.umd.edu
- www.umd.edu
- www.bookwire.com
- ad.doubleclick.net
- me.doubleclick.net
- home.netscape.com
- adforce.imgis.com
- csalpha.unomaha.edu
- www.cs.dartmouth.edu
- messenger.netscape.com
- www.lxpres.com
- tbtf.com
- www.developer.com
- diradserver.developer.com
- flunn.earthweb.com
- www.preemptive.com

Paths:

```
http://java.sun.com/
http://java.sun.com/images/pixel.gif
http://java.sun.com/images/logos/javalogo52x88.gif
http://java.sun.com/images/navbar/fulnav.5.gif
http://java.sun.com/images/banners/pageheader2.2.gif
http://java.sun.com/images/banners/newsandfeatures.divider.gif
http://java.sun.com/images/spotlight.gif
http://java.sun.com/share/classes/menu/horizMenu.class
http://java.sun.com/images/programs.gif
http://java.sun.com/images/logos/jlc.gif
http://java.sun.com/images/banners/products.table.top.gif
http://java.sun.com/images/banners/products.table.bottom.gif
http://java.sun.com/images/rollover/jini.partners.cover.gif
http://java.sun.com/images/navbar/scroll.indicator.v3.gif
```

From Cache: 0

Fresh Pull: 1

Not Found: 0

Other Responses: 0

Total Visits: 36

Last Visit: Tue Feb 02 21:58:13 MST 1999

Figure 38: Enhanced History Assistant Result Window

4.4.6 Persistence

Persistence within ***Sandwich*** is considered to be a framework itself, pertaining to the common service pool referred to in [FHLS99c]. This framework is from the Client Server Framework (CSF) by Garry Froehlich [CSF]. Version 1.1.1 of CSF was investigated during the early stage of this thesis and was adopted. Since then, updated versions of CSF have not been incorporated.

This framework transparently handles storing and loading of data objects within database, files, or any other persistent storage mechanisms. At present, only file support is implemented. A managerial design approach is taken such that all responsibilities to perform storing and loading of persistent data lie within the singleton [GHJV95] object known as the *PersistenceManager*. Like all other CSF core classes, *PersistenceManager* is a subclass of CSF's *CommAwareObject*.

This section elaborates on how this adopted framework has evolved and Figure 39 shows the static class diagram of the evolved persistence framework. This persistence framework was successfully deployed in ***Sandwich*** with minor code changes and additions, while maintaining the fundamental design.

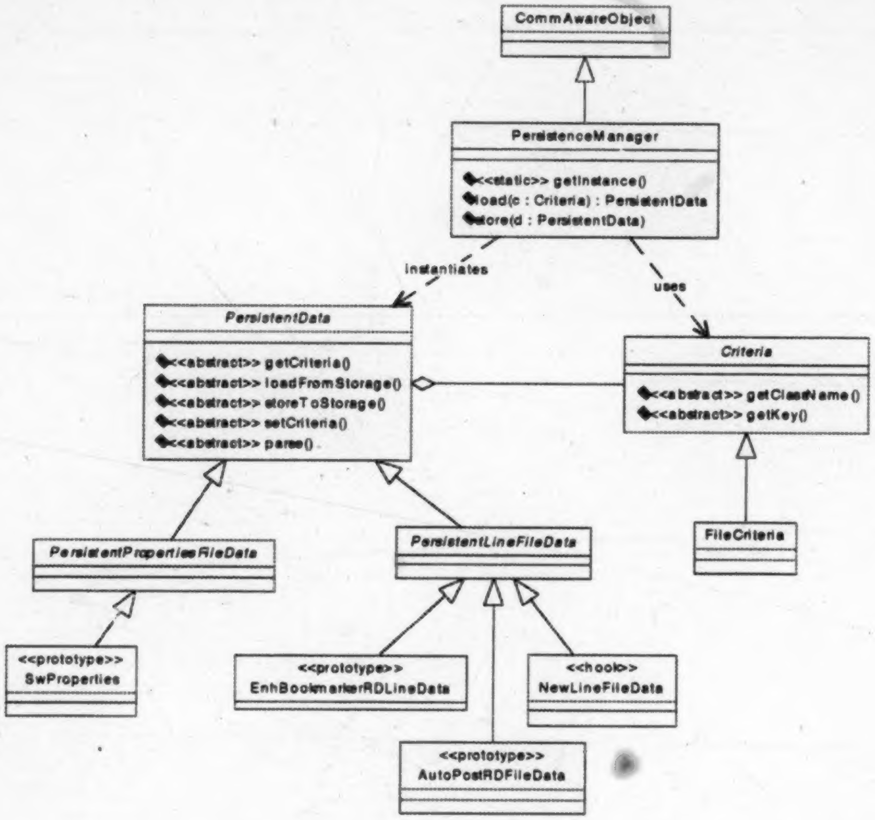

CommAwareObject
PersistenceManager
<<static>> getInstance()
load(c : Criteria) : PersistentData
store(d : PersistentData)
instantiates
uses
PersistentData
<<abstract>> getCriteria()
<<abstract>> loadFromStorage()
<<abstract>> storeToStorage()
<<abstract>> setCriteria()
<<abstract>> parse()
Criteria
<<abstract>> getClassName()
<<abstract>> getKey()
PersistentPropertiesFileData
PersistentLineFileData
FileCriteria
<<prototype>>
SwProperties
<<prototype>>
EnhBookmarkerRDLineData
<<hook>>
NewLineFileData
<<prototype>>
AutoPostRDFileData

Figure 39: Persistence

4.4.6.1 Evolved from CSF v1.1.1.

In CSF v1.1.1 [CSF], the loading and storing of persistent data are based on the following two methods of *PersistenceManager*.

```
Loading:
public Data read (String classname, Criteria c)
Storing:
public void write (String classname, Criteria c, Data d)
```

At the time of adopting this sub-framework, these methods simply delegate to a *FileManager* object to read and write the object whenever necessary. However, the methods that *FileManager* used were only declared and a full implementation was not available for the read and write methods. The following is a summary of changes made to the original CSF v1.1.1 persistence sub-framework with their rationale:

1. A new package named *csf.persistence* is created, and all classes that pertained to the persistence sub-framework are moved there. Originally, all classes of CSF pertained to one big package known as the *csf.* This new package explicitly shows what classes comprised the persistence sub-framework and further enforces low coupling among independent classes through Java packaging visibility rules. This includes the following set of rules:
   (a) only public methods of public classes in package A are visible (i.e. accessible) to any other classes of any other packages other than package A, and
   (b) protected methods of a class are visible to other classes in the same package

2. The methods of *PersistenceManager* were modified to the following:

```
Loading:
public PersistentData load (Criteria c) throws DataReadException

Storing:
public void store (PersistentData d) throws DataWriteException
```

We note the following changes, some of which are further elaborated on below with the indicated item number.

- The method names are renamed from read to load and write to store (mainly due to object naming or style preference),
- The original return type of the load method was *Data*; this is now replaced with the *PersistentData* type (#3),
- The use of class *FileManager* is removed (#4),
- The parameter type of the load method changes from taking a class name and *Criteria* object to only a *Criteria* object (#5); the store method changes from taking a class name, *Criteria* object and *Data* object to only a *PersistentData* object (#3,4),
- The load method was modified to throw *DataReadException*; the store method was modified to throw *DataWriteException* (#6).

- The introduction of two new abstract subclasses of *PersistentData* known as the *PersistentPropertiesFileData* and *PersistentLineFileData* (#7).

3. The previous class *Data* was renamed to *PersistentData* and was made an abstract class. The word "Data" has broad meanings. For example transient data, network serializable data, and persistent data are all considered as data. We felt that *PersistentData* was a better name for representing data that were stored in persistent storage. By making *PersistentData* an abstract class, we ensured that framework users had to provide an implementation for these declared abstract methods.

4. The use of *FileManager* to do the actual reading from and writing to files was removed. This change is mainly due to design style preference. Using the *FileManager* approach is a typical "object manager" approach in accomplishing tasks. We felt that the *PersistentData* object should be responsible for and know how to load and store itself. The latter approach promotes object cohesion.

5. Previously, when a *PersistenceManager* needed to load data, a particular class name and a criteria object must be provided. Through the reflection API of Java, the object for the given class name is instantiated and then assigned the criteria. The evolved version moved this class name value into the criteria object itself. This seems appropriate as a criterion often applies a set of conditions. We felt that the class name was one of these conditions and thus could be encapsulated in the *Criteria* object itself. Additionally, the class *Criteria* was now abstract to further ensure that the framework user provided an implementation for its abstract methods.

6. Exception handling was added because there were obviously occasional times when persistent data cannot be read or stored properly. This could be due to input/output error while reading from or writing to file or invalid instantiation of *PersistentData* through the reflection API.

7. Two new subclasses of *PersistentData* are added.
   - Class *PersistentPropertiesData* defines the *loadFromStorage ()* abstract method to read the file content into a JDK *Properties* object. This is done to facilitate property files that contain a set of lines in the form of key1=value1.

- Class *PersistentLineFileData* defines the *loadFromStorage ()* abstract method to read in a file into a vector of lines as strings. Both of these new subclasses are abstract as thus defers the implementation of the *parse()* method to its subclasses, see template method [GHJV95].

Name	Reading Property File Hook
Requirement	Read a property file into a *SwProperties* object.
Type	Enabling Pattern
Area	Utilities
Participants	*FileCriteria* aFileCriteria, String aPropertyFileName, *SwProperties* aSwProperties
Uses	None
Pre-conditions	None
Changes	1. code: aFileCriteria = new FileCriteria ("com.config.SwProperties", aPropertyFileName) 2. code: PersistentManager mgr = PersistentManager.getInstance(); 3. code: aSwProperties = mgr.load(aFileCriteria);
Post-conditions	A new *SwProperties* object is initialized with the content of aPropertyFileName. Thus, all property values can now be obtained using this *SwProperties* object.
Comments	None

Hook 6: Reading Property File Hook

During the early stage of ***Sandwich*** construction, we had a **New Properties File Data Hook**. However, during the final revisit, this hook was removed due to its redundancy. This is mainly because properties files format is well defined, and there is no additional business logic to be interpreted in the template [GHJV95] method *parse()*. In other words, the **Reading Property File Hook** suffices in most cases for a given property file.

Following are two other hooks written for the extension added to the persistence framework, specifically for supporting line file data as the persistent storage.

Name	New Line File Data Hook
Requirement	Encapsulating the content of a particular non-property file as a *PersistentLineFileData* object.
Type	Adding Pattern
Area	Utilities
Participants	Abstract class *PersistentLineFileData*. String aLineFileName
Uses	Reading Line File Hook
Pre-conditions	None

Changes	1. new subclass NewLineFileData of PersistentLineFileData
	// construct the appropriate business objects from the file
	// aLineFileName
	2. NewLineFileData.parse() overrides PersistentLineFileData.parse()

Post-conditions	A new subclass of *PersistentLineFileData* that knows how to parse the content of aLineFileName is created.
Comments	None

Hook 7: New Line File Data Hook

Name	Reading Line File Hook
Requirement	Read a file into a *PersistentLineFileData* object.
Type	Enabling Pattern
Area	Utilities
Participants	*FileCriteria* aFileCriteria, String aLineFileName, aPersistentLineFileData
Uses	None
Pre-conditions	There is a concrete subclass of *PersistentLineFileData*.
Changes	1. Note: Declare a variable className of String type and assigned it the full class name of a *PersistentLineFileData* subclass 2. Code: aFileCriteria = new FileCriteria (className, aLineFileName) 3. Code: PersistentManager mgr = PersistentManager.getInstance(); 4. Code: className aPersistentLineFileData = mgr.load(aFileCriteria);
Post-conditions	A *PersistentLineFileData* object is initialized and loaded with the content of

	aLineFileName. This object is an instance of a *PersistentLineFileData* subclass mentioned in pre-conditions.
Comments	To create a concrete subclass of *PersistentLineFileData*, see the **New Line File Data Hook.**

Hook 8: Reading Line File Hook

4.4.7 Logging

Logging is an important aspect of all software systems. A typical logging system allows for more effective trouble shooting by developers when the user encounters a problem at run time. At present, the logging support of ***Sandwich*** is minimal but can be very easily extended.

Sandwich has declared an abstract class *Log* for logging different types of messages such as debugging, informational, warning, and error messages, see Figure 40. This *Log* class has a *public static getLogger()* method that creates and returns a singleton *Log* object. Thus, this class plays the role of a singleton as well as a factory [GHJV95]. Since *Log* is an abstract class, its implementation must be provided by one of its subclasses. The chosen subclass or implementation is plugged in as the *Log* object at run time using static initialization and the *protected void setLogger* method. Stereotyped classes included in Figure 40 include (a) default logger, *LoggerStdout*, and (b) the hot spot of the new logger hook.

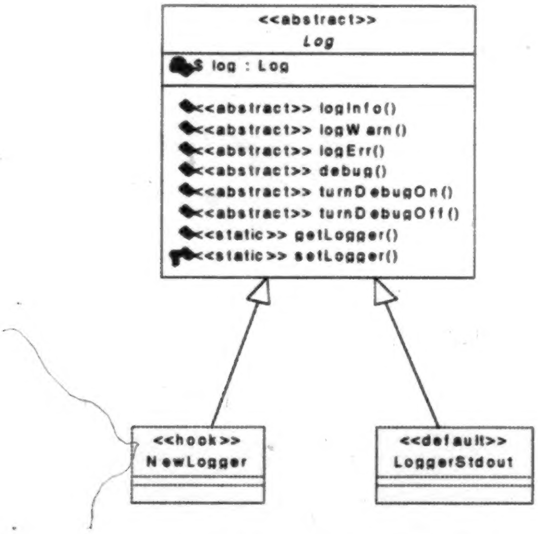


Figure 40: Logging Support

Different subclasses of *Log* provide different implementations of logging; the default implementation is through the subclass *LoggerStdout* that simply prints messages to standard output. The creation of the desired *Log* object or logger is a hot spot in ***Sandwich***. Thus, another logger can be easily hooked in by the user without any code change required in the framework. The framework user may choose to implement new subclasses for more robust and sophisticated implementation, such as one that logs messages to files or database tables, or one that supports asynchronous logging. The **New Logger Hook** below describes how the steps in creating new logger and replacing the default logger with the new one. All existing messages that are currently logged in the framework will then be logged through this new logger without any additional code change in the framework. This lies on the fact that the framework is always and only needed to be aware of the abstract *Log* class.

Name	New Logger Hook
Requirement	A more robust logging mechanism is desired.
Type	Replacing Pattern
Area	Utilities
Participants	Abstract class *Log*, the *sandwich.properties* file
Uses	None
Pre-conditions	None

Changes	1. new subclass NewLogger of Log
	// define all the abstract methods of the Log
	2. NewLogger.logInfo(String) overrides Log.logInfo(String)
	3. NewLogger.logWarn(String) overrides Log.logWarn(String)
	4. NewLogger.logErr(String) overrides Log.logErr(String)
	5. NewLogger.debug(String) overrides Log.debug(String)
	6. NewLogger.turnDebugOn() overrides Log.turnDebugOn()
	7. NewLogger.turnDebugOff() overrides Log.turnDebugOff()
	8. note: includes the following static initialize code in NewLogger
	`static {`
	`setLog( new NewLogger() );`
	`}`
	9. note: update the configurable parameter that tells the framework which logger to use in the sandwich.properties file:

	• logger_classname=NewLogger's className
Post-conditions	NewLogger will be used instead of the default logger
Comments	None.

Hook 9: New Logger Hook

4.4.8 Regular Expression

Recall that an observing assistant specifies which data element (in particular which HTTP headers) it is interested in observing. Adding regular expressions to HTTP headers allows the framework to send out notification only when interested data changes occur.

For example, with the default regular expression built in, the following is true.

```
Synopsis:
          <header name>,<regular expression>
# E.g. Framework will notify observing assistants that are interested in response code
# that is one of 200, 304, or 404.
          com.http.HttpResponseCode,200|304|404
```

Classes involved in supporting regular expression in ***Sandwich*** are shown in Figure 41. To reduce coupling between ***Sandwich*** and any third party library that is used to support the regular expression implementation, a factory class and an interface are created and these are the only classes/interfaces that ***Sandwich*** is aware of. Optionally the adapter (wrapper) pattern [GHJV95] is used. To elaborate,

(a) the *RegExpFactory* factory class is responsible to create the appropriate regular expression implementation object,

(b) the *RegularExpressionIF* interface that declares the required methods by the framework,

(c) the adapter pattern allows incompatible interfaces to work together, thus this pattern can be applied when third party classes that support regular expression are reused in ***Sandwich***. In other words, those third party classes are adapted to the *RegularExpressIF* expected by ***Sandwich***. The default *FastMatcherAdapter* class wraps the third party classes from a free Java regular expression package (http://www.cs.umd.edu/users/dfs/java/). This same technique can be applied should a different regular expression library be picked over this default one.

Stereotyped classes included in Figure 41 include

(a) class *FastMatcherAdpater* that is the present default regular expression support included in ***Sandwich***, and

(b) class *NewRegExpressionAdapter* shows where the new class will be added when enacting the **New Regular Expression Adapter Hook** described below

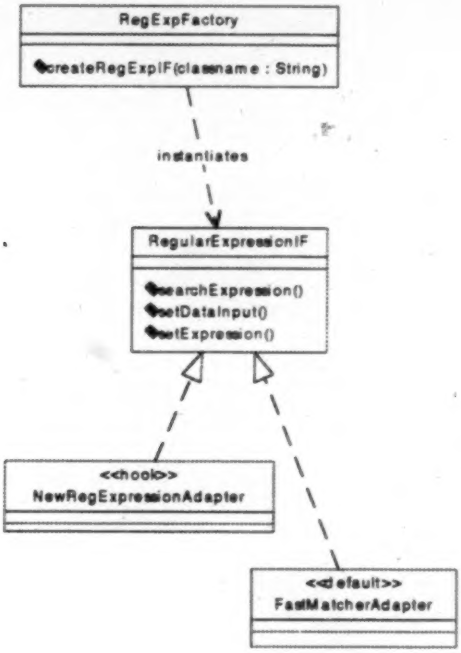

RegExpFactory
createRegExpIF(classname : String)
instantiates
RegularExpressionIF
searchExpression()
setDataInput()
setExpression()
<<hook>>
NewRegExpressionAdapter
<<default>>
FastMatcherAdapter

Figure 41: Regular Expression

Name	New Regular Expression Adapter Hook
Requirement	Required another set of regular expression, probably due to some deficiency of the default regular expression hooked in
Type	Replacing Pattern
Area	Utilities
Participants	*RegularExpressionIF* interface and the *sandwich.properties* file
Uses	None
Pre-conditions	None
Changes	1. new class NewRegExpressionAdapter implements *RegularExpressionIF* 2. note: update the *sandwich.properties* file: regular_expression_classname=NewRegExpressionAdapter class name
Post-conditions	The next time ***Sandwich*** is started, the NewRegExpressionAdapter will be used instead of the default FastMatcherAdapter

Comments	• NewRegExpressionAdapter is likely to contain or inherit a third party class that support the desirable regular expression

Hook 10: New Regular Expression Adapter Hook

4.4.9 HTTP Support

The HTTP protocol will continue to evolve over the next few years as the web develops. As the specification changes, ***Sandwich*** implementation must be revisited. If new headers are added, the following **New HTTP Header Hook** can be applied to evolve ***Sandwich*** such that this new header is supported.

Currently, not all headers defined in HTTP 1.1 are implemented in ***Sandwich***. Only those that are directly used by one of the prototypes are coded and tested thoroughly. It is very easy to add support for all other headers with the directions outlined in the **New HTTP Header Hook**. Figure 42 shows the static relationships of all *HttpHeader* related classes, in particular

- Stereotyped class *NewHttpHeader* shows where the new class will be added when enacting the **New HTTP Header Hook**.
- Collaboration between *HttpHeader* and *ObservingIF* exist to support the notification to observing assistants when the state of the data element of the *HttpHeader* objects changes.
- All *HttpHeader* classes are subclasses of the abstract *HttpHeader* class.

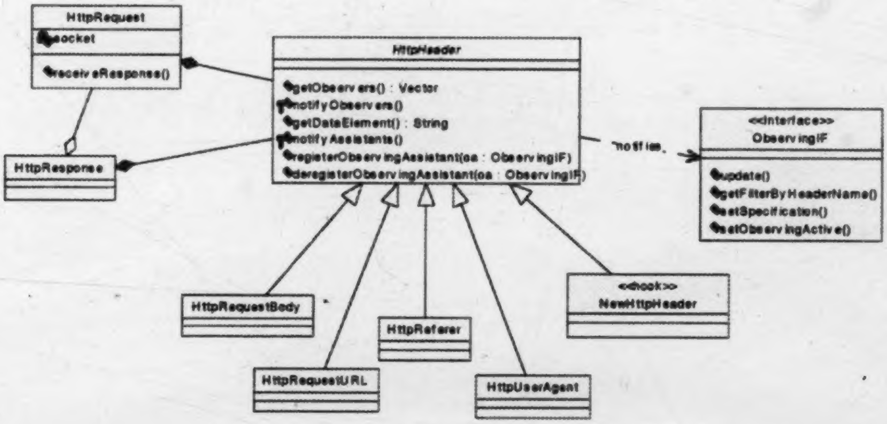

HttpRequest
socket
receiveResponse()
HttpResponse
HttpHeader
getObservers() : Vector
notifyObservers()
getDataElement() : String
notifyAssistants()
registerObservingAssistant(oa : ObservingIF)
deregisterObservingAssistant(oa : ObservingIF)
notifies
<<interface>>
ObservingIF
update()
getFilterByHeaderName()
setSpecification()
setObservingActive()
HttpRequestBody
HttpRequestURL
HttpReferer
HttpUserAgent
<<hook>>
NewHttpHeader

Figure 42: HTTP Support

Name	New HTTP Header Hook
Requirement	An additional HTTP Header needs to be supported and thus becomes available for observing assistant to register against.
Type	Adding Open-Ended
Area	Utilities
Participants	*HttpHeader* abstract class
Uses	None
Pre-conditions	This new HTTP Header is not yet supported in ***Sandwich***.

Changes

```
1.  new subclass NewHttpHeader of HttpHeader
    // declare new variable for keeping the data element of
    // NewHttpHeader type is typically String but depends on the header,
    // for example, HttpDate might use type Date instead
2.  property newDataElement
3.  NewHttpHeader.getDataElement() overrides
    HttpHeader.getDataElement()
        returns newDataElement
    // declare new static variable for a list or vector to store registered
    // observers
4.  new class property NewHttpHeader.list of type list
5.  NewHttpHeader.getObservers() overrides HttpHeader.getObservers()
        returns NewHttpHeader.list
6.  note: when the date element state changes or is set initially (usually
    encapsulated in the setter method of the data element), this
    NewHttpHeader object should invoke its superclass's
    notifyAssistants(Object) method
```

Post-conditions	This new HTTP Header is now available for registration by observing assistants.
Comments	• This hook can be applied when a new HTTP header is introduced as the HTTP protocol evolves or as the framework evolves to support more HTTP headers. • The data element of the HTTP header is often the value of the HTTP header, recall that each header is a name-value pair.

Hook 11: New HTTP Header Hook

Why does the *HttpHeader's* register and deregister observing assistant take a parameter type of *ObservingIF* rather than *Assistant*?

The Law of Demeter says that if two classes have no reason to be directly aware of each other then they should not directly collaborate [Grand99]. According to this rule of thumb and because *HttpHeader* objects are only responsible to notify observing assistants, there is no reason for *HttpHeader* to interact with *Assistant* directly but with *ObservingIF* instead. This is why the *registerObservingAssistant* and *deregisterObservingAssistant* both take a parameter of type *ObservingIF* rather than the type *Assistant*.

Why is the New Http Header Hook considered to have the open-ended level of support?

This hook provides directions to extend the framework itself. The user of this hook will have to possess a deep knowledge of the framework design and the HTTP protocol in order for them to determine when to notify its observers based on the state change of its data element. This maps to having the highest level of support, open-ended, in the hook model.

4.5. Steps In Creating New Sandwich Assistant

This section summarizes the steps required in identifying and constructing a new assistant using ***Sandwich***:

1. There is a new requirement for a new assistant.
2. Use case *Identify an Assistant* must first be realized, i.e. an application use case for the new assistant has been written and the developer has verified that the requirement of this new assistant can be fulfilled in ***Sandwich***.
3. Determine if the assistant needs to monitor the user browsing HTTP stream. If so, identify which HTTP header(s) the assistant is interested in. If ***Sandwich*** does not yet support the HTTP header that this assistant is interested in, enact the **New HTTP Header Hook** for the new header.
4. Determine if the assistant delegates on behalf of the user. If so, determine whether the assistant is a request or response delegate.

   If the assistant meets the following criteria, then the assistant is likely to be a request delegate:

   - needs to modify the incoming request
   - needs to create a new response for a particular request

   If the assistant meets the following criteria, then the assistant is likely to be a response delegate:

- needs to change the content of an existing response
- needs to create a new response for a particular response

5. If the assistant is a delegating type, then determine the priority of it.
6. Create the assistant using the corresponding hook, e.g. **New Observing Assistant Hook** for step 3, and the **New Request** or **Response Assistant Hook** for step 4
7. Assuming the default administrative application is used, if the assistant requires some GUI to the end user, then the **New Assistant Result Window Hook** might be applied.

4.6. Other Considerations

4.6.1 Assistant that is Observing and Delegating

Figure 43 shows an example where the assistant is both an observing and a delegating type. More specifically, the assistant is a request delegate; as an observing assistant, all states of *HttpHeaderA* are interested. Due to the nature of asynchronous approach that is deployed in supporting observing assistants, it is possible for this assistant to get notified by *HttpHeaderA* of service thread 2 when the assistant is servicing a *HttpRequest* from service thread 1. Thus, assistant that plays the role of observing and delegating will likely need to know which notification of *HttpHeaderA* corresponds to the current request or response object that it is servicing. Using the thread number of *HttpServiceThread* that is responsible in fulfilling the current *HttpRequest*, this requirement can be easily fulfilled. The details of this, in terms of data structures, were included in Section 4.4.3 in the discussion of *AssistantSpecification* containment for *Assistant*.

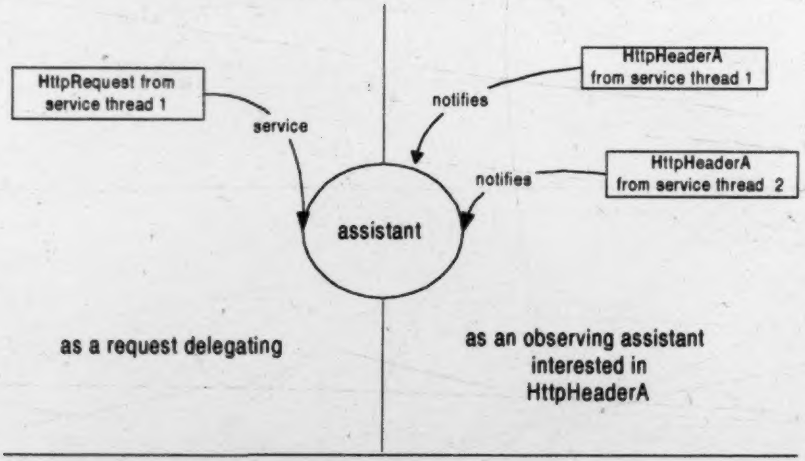

HttpRequest from service thread 1
service
HttpHeaderA from service thread 1
notifies
HttpHeaderA from service thread 2
notifies
assistant
as a request delegating
as an observing assistant interested in HttpHeaderA

Figure 43: Assistant that is both Observing and Delegating

4.6.2 Request and Response Delegates Pair

There will be occasions that a request delegate needs to collaborate with a corresponding response delegate and vice versa. More generally, as the scope of assistants that are to be supported expands, there will likely be a need for an assistant-to-assistant collaboration. Currently, we do not know the best approach to accomplish this goal. A study of protocols for assistant-to-assistant communication is required, in conjunction with the development of prototypical collaboration assistants.

In the current state of ***Sandwich***, the collaboration among two or more assistants can be accomplished through a simple mechanism such as direct methods invocation as described below using an example, see Figure 44. Assumed that *RequestDelegateA* is interested

- to perform delegation when the requested URL is Fred URL,
- to use *ResponseDelegateB* for the response generated by its delegation

To accomplish this today with ***Sandwich***, during the execution of the *service()* method of *RequestDelegateA* for *HttpRequest* with Fred URL, the following occurs:

1. *RequestDelegateA* performs its delegation as usual and produces a *HttpResponse* for Fred's *HttpRequest*.
2. *RequestDelegateA* gets the object reference to *ResponseDelegateB* from the proxy.
3. R*equestDelegateA* invokes the *service()* method of *ResponseDelegateB*, giving it the *HttpResponse* object from the above step.
4. *ResponseDelegateB* performs its delegation as usual and outputs a *HttpResponse* object, which is returned to *RequestDelegateA*.
5. *RequestDelegateA* returns this *HttpResponse* as the result of its *service()* method

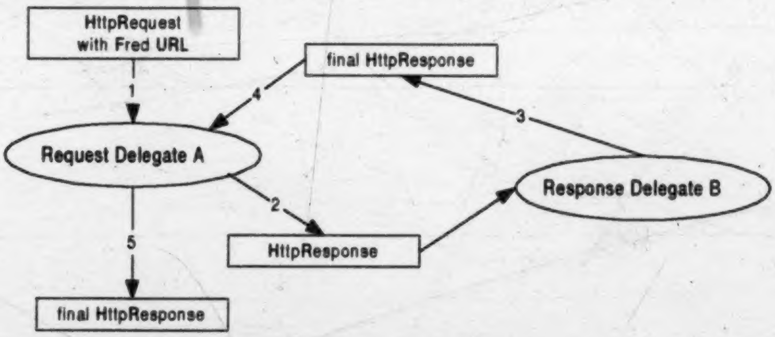


Figure 44: Request and Response Delegates Pair

5. Evaluation

During the first iteration of ***Sandwich***'s construction, a quick prototype – a rudimentary Java-based HTTP proxy server – was built. With the knowledge gained from this prototype, we investigated what sort of assistantships are candidates of the frameworks, what are the common services that can be shared by assistants and how this framework should support these common services through its hot spots. The next or second iteration involved a rapid prototyping of an observing assistant example, the Enhanced History Assistant. This provided a proof of concept for the framework design to support observing assistants. The last or third iteration applied the hook model, involved some class re-factoring and developed a delegating assistant known as the Check Free Auto Post Assistant. Similar to the previous prototype, this assistant was another proof of concept for the framework design, only this time it supported a delegating assistant.

In this chapter, we first evaluate the major techniques and tools that are used during the framework construction and documentation. These include (1) the hook model used to document the framework extension points, and (2) UML used as the notation standard in use cases and object diagrams. We then provide a comparative analysis between ***Sandwich*** and Webby, as promised in Section 2.3.3.

5.1. The Hook Model

The hook model is used to document the extensions of ***Sandwich***. The hook model was successfully applied during the last iteration of the framework construction. The reason for not being able to use the hook model during the first two iterations was mainly due to the unfamiliarity of the solution approach, e.g. frameworks in general, the Java programming language, and the instability of the framework design. The last or third iteration provided the most complete and stabilized design and therefore we felt that applying the hook model to this last iteration was appropriate. The following summarized the lesson learned in applying the hook model in this thesis.

IMPROVED FRAMEWORK USAGE

By using the hook model, the framework designers can identify where the hot spots of the systems are and the steps on how to extend the framework in a more formalized manner. We found that if the hook for a particular hot spot could not be easily written, then the soundness of the design around the hot spot should be questioned.

AREA OF HOOKS

The current mechanism to group hooks together is through the "Area" field of the hook template. No clear guidelines are documented on how framework developers should define their "Areas." This can be further improved. At present, rather vague or intuitive guidelines are that the "Area" field value is based on (a) the commonality of the services that the hook provides, or (b) subsystem components of the whole system. Meanwhile, some areas can be predefined in the hook definition, and framework developers can add more domain-specific areas as needed. An unanswered question is what other gains are there in having the "Area" field in addition to simply organizing hooks. A misconception that we had initially was that hooks with the same "Area" values are to be enacted altogether.

SUPPORT FOR PROPERTY CHANGES

Based on the hook model, a property change that can turn on/off a feature is considered as an optional level of hook. The hook language addressed well property changes that applied to objects, however, lacked the syntax to illustrate how to make changes to a system property that is stored in a property file. Maintaining persistent properties is an important aspect of all software systems, and thus we feel that a more stringent syntax to support them in the hook model is desirable in the future.

REQUIREMENT OF HOOKS

In the hook template, there is a field for "Requirement" as "a textual description of the problem the hook is intended to help solve. The framework builder anticipates the requirements that an application will have and describes hooks for those requirements" [FHLS99a]. There are no strict guidelines as to how the value of this field maps to the requirement specifications of the framework. We feel that there is a close relationship between this "Requirement" field and anticipated use cases that are typical in the application domain. Further analysis in determining this relationship might be worthwhile.

HOOK TARGETED USERS

Depending on the skills of the framework developers and users, the hook model may appear to be very easy or extremely sophisticated. It may be worthwhile to indicate the sort of user experiences that the hook model is most suitable for. Initially, when we were first introduced to the hook model, in order to understand it, we required real code examples that have enacted the hooks. After a detail study of few hooks, the hook model appears to be much easier to understand. Perhaps, the hook model can be constructed in different granularity, with each level targeting different types of users, based on their knowledge of object technology.

EXAMPLES OF HOOKS

SEAF [FHLS99a] and CSF [CSF] are the other two projects that include hook documentation. Most of these examples are enabling open hooks with a couple of adding open and one enabling option hooks. Through our experience in this thesis, in the context of documenting the framework with hooks, we got started by learning to write hooks through the existing hook examples. Improving the pool of hook examples, covering all possible hook types, will further promote the use of hook model.

LANGUAGE DEPENDENCY

The current hook model has its own grammar definition [FHLS98a] and is programming-language independent. This is a nice approach because

(1) It allows the building of a graphical hook tool to be language independent. Luyuan Liu, another SERL graduate student, is currently investigated this tool for his Masters thesis [Liu99].

(2) It facilitates the integration of multiple frameworks built in different languages. Assuming all framework extensions are specified with the hook model, the communication gap among integrators will be minimized as they share the common terminology of the hook model.

Nonetheless, there are a few drawbacks to being language independent. First, there is yet another need for both framework developers and framework users to learn and understand another new language. Also, amateur programmers who are not familiar to the hook model and the framework's implementation language might not be able to quickly enact a hook. Translation guidelines for each popular OO programming language (e.g. Java and C++) to reduce this overhead are desirable.

DEPENDENCIES AND OPTIONAL PATHS

A simple scenario where hook dependencies occur is when a particular hook cannot be enacted by itself but requires the use of some other additional hook(s). For example, in Figure 45 (a) Hook A requires the use of Hook B. The "Uses" field of the current hook model addresses this scenario sufficiently.

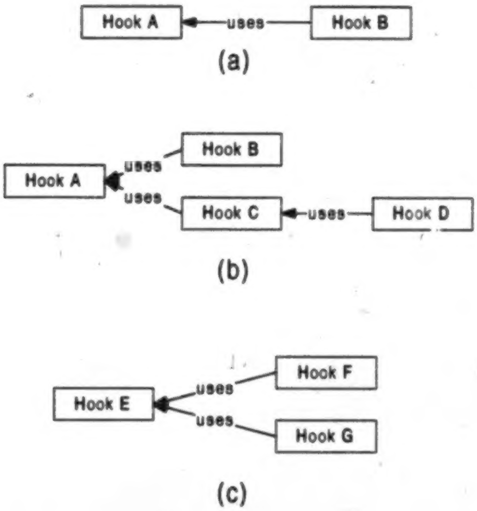
Optional paths occur when various combinations of hooks can fulfill a certain requirement. This happens regularly in a framework where a hot spot can have various hooks. A generic example is illustrated in Figure 45 (b) where a certain requirement can be fulfilled either with

- Hook A that uses Hook B, or
- Hook A that uses Hook C that in turn uses Hook D

The latest grammar of "Uses" is

<uses> ::= Uses: <hook name> [, …, <hook name>]

There is the lack the capability to show this optional path scenario. Adding an "|" for "OR" might be the easiest modification to address this scenario.



(a)

(b)

(c)

Figure 45: Hook Dependencies and Optional Paths

A more complex scenario occurs when both hook dependencies and optional paths exist together for the same requirement. For instance, in Figure 45 (c), depending on which path of hook E we choose we must use a particular corresponding hook A path. In particular, a certain requirement can be fulfilled with either

- Hook E that uses F and Hook A that uses B, or
- Hook E that uses G and Hook A that uses Hook C and D

This scenario is tackled well in the improved version of the hook model that includes the "Pre-Conditions" and "Post-Conditions" fields. The grammar for these two fields is still in progress.

Looking back on this section and how a complicated situation can easily appear, we feel that additional pictorial illustration, especially for complex hook dependencies and optional paths, to the hook model is necessary.

HOOK ATOMICITY

A particular hook is considered as atomic if it can be used by itself. The notion of atomicity does not yet exist in the proposed hook model. We feel that there is a need for it. For example, when creating new assistant in ***Sandwich***, the user must choose from the three hooks:

- **New Observing Assistant Hook**,
- **New Request Delegate Assistant Hook**, and
- **New Response Delegate Assistant Hook**

In theory, there can be a generic hook for creating new assistant without determining the type of the assistant. This generic hook can have a name like the **New Assistant Hook** and can be used by the above three hooks. However, this hook cannot be enacted by itself (non-atomic) because ***Sandwich*** does not consider an assistant that does not observe or delegate as an assistant.

Based on the current hook model, the **New Assistant Hook** is not considered to be a hook and the "changes" section of the hook is to be repeated for all the three above hooks. For this particular example, the following lines that are anticipated to be in the "changes" instruction of **New Assistant Hook** have been repeated in the "changes" instruction of the above three hooks.

```
new class NewAssistant subclass Assistant
   // to perform any assistant-specific
   // initialization
   •   NewAssistant.load() overrides Assistant.load()
   // to release any system-resources held
   •   NewAssistant.finish() overrides Assistant.finish()
```

Thus, if the "changes" instructions of non-atomic hook are huge, repeating them to every hook that uses it can potentially produce redundancy, unnecessary inconsistency, and mistakes. The simplest approach to overcome this might be introducing a new field into hook template that indicates whether a hook is atomic or not. This then allows non-atomic hooks to be extracted out. Other hooks that use it can then indicate their dependencies on it using the "Uses" field.

COMPLEMENTS TO HOOK DOCUMENATION

At present, what constitutes good framework documentation is still an open research question. Needless to say, the hook model will not be alone and thus will co-exist with some other sections of framework documentation. To promote the integration of the hooks to other sections of the document and improve better understanding of the hooks, we recommend the inclusion of the following non-exhaustive list for the stated reasons:

1. Use cases. These can be directly referred to in the "Requirement" field of hook description. This could further improve the context of the hook applicability.
2. Some sort of subsystem model that illustrates the major services offered by the framework. This can directly map to the "Area" field of the hook description.
3. With option hooks, static class diagrams on the objects involved suffice. With pattern and open-ended hooks, we feel the need to have dynamic models of the objects involved. This is because for the latter types of hooks, users are expected to know more about the framework dynamics, and having collaboration or sequence diagrams together with certainly help.
4. An on-line documentation system with the use of hypertext. This allows the linking together of related hooks as well as their corresponding examples. From the experimental studies conducted by Garry Froehlich with CMPUT 401, an undergraduate software-engineering course at the University of Alberta, students were more comfortable when code examples explaining how to use the hooks were also provided.

We conclude that the hook model is reasonably good at formalizing the documentation for how a framework can be extended. It has the potential to show precisely where the hot spots are for the given frameworks and what the hooks are for these hot spots. With the on-going work to add diagrammatic notation to the hook model, the potential becomes more appealing. Nevertheless, there is a cost to using the hook model. For both the framework developers and users, there is the overhead of learning and adopting the hook model with its own set of grammar and templates. Also, it is important to remember that the hook model is not the only form of framework documentation. It is also recommended to complement hook documentation with easy-to-understand examples constructed using the framework by enacting the hooks.

5.2. Applying UML

This section summarizes the experience gained when applying UML to the Object-Oriented Analysis and Design (OOAD) phase of ***Sandwich*** construction. General observations and particular features of UML that are deployed in this thesis will be commented on.

1. UML is perhaps the best notation standard available in documenting frameworks. UML is becoming more widely accepted by a large user community. Using it will promote the understanding of a framework to a greater audience.

2. UML should be used carefully. It is important for developers to be aware that UML is a notation standard for a diagramming language. UML helps in the OOAD modeling phase of software development by improving communication and understanding among modelers via diagrams with standard notations. Often, there is a misconception that people with skills to read and write UML can model. The latter is a much more important skill; one can be a good modeler without knowing UML. Some background is desired when deploying UML to ensure that developers use it correctly and efficiently and not have their models restricted by any UML shortcomings.

3. Rational Rose 98 is a good tool that provides intuitive interface in drawing UML diagrams. Rose is easy to learn and simple to use.

4. Static class diagrams are used to capture the objects of the system, in the form of relationships, class hierarchies, and dependencies. We found that as the number of classes grows, putting all classes together and showing their static relationship is impossible. Thus, we took the approach of grouping classes based on their core services. For example, only classes that are directly involved in logging will be shown on the logging static class diagram. This improves readability and organization of classes that collaborate or are related.

5. Static class diagrams can also be tagged as *stereotypes*. This feature of UML allows us to diagrammatically show three important concepts (prototype, hook and default) related to a framework. As mentioned before, another active research [Liu99] is currently undergoing in SERL in modeling hooks using the UML's *project* notation.

6. Use cases are used to capture requirements of ***Sandwich***. We found that use cases is very useful when capturing the functional requirements of the targeted applications. The exercise of writing the use cases for the set of representative applications has helped us to capture not only the common functionality of the applications that the framework should support but also the flow of control that should be embedded in the framework. On the other hand, we found

that deriving the use case for the framework by generalizing the applications' use case was not a very helpful exercise. We felt that framework use cases are too abstract to be meaningful although, by nature, framework use cases are supposed to be generalization of the applications' use cases. More studies need to be conducted in this area of software engineering in terms of the guidelines or protocol in writing use cases for frameworks and determining their usefulness.

We conclude that some parts of UML are good enough to capture framework models; others are not. The use case component of UML is very helpful in capturing the requirements of the targeted applications. In particular, the set of use cases written for the chosen set of applications have facilitated us in capturing the common functionality of the applications and the control flow that the framework should support. However, due to the abstract nature of framework use cases, we found that writing use cases for the framework has produces less benefit. Static class diagrams are sufficient when applying them to the OOD (Object-Oriented Design) phase of framework development. UML's *stereotype* provided additional flexibility that is useful in capturing framework design concepts.

5.3. *Sandwich* versus Webby

During the initial startup period of ***Sandwich***, a copy of Webby was downloaded and evaluated. At that time, the download was not runnable, and there was only a limited amount of documentation. In particular, there was no documentation for how to add new assistantship to Webby. This copy of Webby, referred to as WBI Application, is similar to the applications WebMate and WebeW (Section 2.3.3). The WBI Application provides personal web assistantship on personal history and traffic lights. This copy of Webby is being referred to as the WBI Application and was originally written in Perl (1996) and later ported to C++ and finally Java in 1997. The Webby team is comprised of about 2-3 full-time IBM Almaden research staff.

During the wrap up phase of ***Sandwich***, we revisited the IBM alphaWorks page and downloaded the first official (June 1999) release of Webby. This time it was referred to as the WBI Developer Kit and not the WBI Application. To our surprise, we found that its demo programs run, and there is documentation for how to add new assistantship. Below, when the word "Webby" is used, WBI Developer Kit is implied.

Like ***Sandwich***, Webby's approach to provide a personal web assistant framework is also through an intermediate HTTP proxy. Users of Webby can add more assistantship by writing and adding new "plug-ins." The documentation of Webby, however, does not claim it to be a framework but "a programmable HTTP request and response processor." Webby's initial installation came with a pool of five plug-in: Personal History, Traffic Lights, PageFilter, Yahoo Subjects, and XML/XSL. Another important similarity is that both ***Sandwich*** and Webby assumed that the end user trusts the intermediate HTTP proxy fully.

In Webby, an HTTP stream passes through one or more registered plug-ins, each plug-in consisting of one or more of the following, referred as MEG:

(1) A Request Editor (RE) that can optionally change the request,
(2) A Generator (G) that takes a request and outputs a response,
(3) A Document Editor (DE) that can optionally change the response, and
(4) A Monitor (M) that can be designated to receive a copy of request and response but cannot otherwise intercept with the stream flow

Initially, a Webby's "plug-in" sounds like the counterpart of ***Sandwich***'s "assistant." After a more thorough study on what comprises a Webby plug-in, we see that a Webby's MEG element is the counterpart of a ***Sandwich***'s assistant "role". Thus, an assistant that plays two roles, such as observing and request delegating, is the counterpart of a Webby plug-in that consists of two MEGS, Monitor and Request Editor.

The following summarizes the counterparts of each of ***Sandwich***'s major features. An asterisk in column one indicates that a numbered comment on this feature follows Table 4.

	Sandwich	Webby
[1]	Overall control flow: • request delegates are traversed first followed by response delegates • observing assistants cannot intercept the control flow; they simply receive event notifications from interested HTTP headers that fulfill the regular	Overall control flow: • request editors are traversed first followed by generators and then document editor • monitor cannot intercept the control flow, they receive the request and response objects that they are

	expression set	interested based on the conditions set
[2]	Use of intermediary proxy as the basis.	Use of intermediary proxy as the basis.
[3]*	Request Delegating "role" - *Input*: request - *Output*: request', response	a) Request Editor MEG, and - *Input*: request - *Output*: request' b) Generator - *Input*: request - *Output*: document (response equivalent)
[4]	Response Delegating "role" - *Input*: response - *Output*: response	Document Editor MEG - *Input*: document - *Output*: document
[5]*	Observing "role" - *Input*: HTTP headers notification - *Output*: result panel upon request	Monitor MEG - *Input*: request, document - *Output*: HTML pages (often) upon request
[6]	Assistant that plays multiple roles, such as observing and request delegating	Plug-in that comprises of multiple MEGS such as Monitor, Request Editor and Generator
[7]	Default request delegating assistant, *DirectHttpAssistant*	Default Generator MEG
[8].	Delegating assistants have priorities set - use of priority queues	Request Editor, Generator and Document Editor MEGS all have priorities - use of priority queues
[9]	Support for regular expression for observing assistant - To change the regular expression of an assistant, change the property file and restarts ***Sandwich***	Support for set conditions for each MEGs. - To make a condition change for a MEG, need to change the source code, recompile and rerun.
[10]	Minimal administrative support but can be easily built on top of current default administrative application	Administrative support through the console window where Webby was started.
[11]	Registering new assistant involves	Registering new plug-in involves the use of

	making new entries to the property file and restarting ***Sandwich***.	the keyword "register" in the command prompt of the administrative console window. Plug-in is registered without restarting Webby.
[12]*	Documentation includes • overview • use cases for the applications • use cases for the framework • architecture • design and its rationale using UML static and sequence diagrams • hooks • assistant examples	Documentation includes • overview • architecture • programming • API • FAQ • plug-in examples

Table 4: *Sandwich* and Webby

[3] Webby has broken down the role of request delegating into two MEGS based on the possible return type. The input type to request delegate and the two MEGS (Monitor and Request Editor) is the same, a request object. ***Sandwich***'s request delegate outputs either a modified request or a response; Webby's Request Editor MEG outputs a request' and Generator MEG outputs a response. Thus, the role of a request delegating can be accomplished with a Request Editor and Generator MEGS pair in Webby.

[5] The design of Webby's Monitor is slightly different from ***Sandwich***'s observing assistant in two perspectives: processing and presentation.

Processing Perspective: how the input of observing assistant or Monitor MEG is supported.

- In ***Sandwich***, an observing assistant indicates which HTTP data elements that it is interested in, along with a regular expression for each of these HTTP data elements. When such a data element exists in an HTTP stream, the assistant is notified by an event. This event notification approach is an asynchronous approach similar to those used in MVC [KP88] or Observer pattern [GHJV95].

- In Webby, a Monitor MEG expresses its interest in an HTTP stream with a condition rule. When this rule is true for an HTTP stream, all information in the stream is passed into the monitor synchronously.

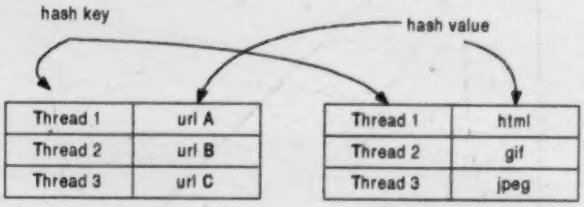
The two approaches are quite different in the following aspects:

- ***Sandwich*** supported an asynchronous way in informing observing assistants; Webby supported a synchronous way in informing Monitor MEGS
- ***Sandwich***'s observing assistants expressed their interest on a per HTTP header basis; Webby's Monitor MEGS expressed their interest on a per-request or per-response object basis. Note that in the ***Sandwich*** model, a request and response object contains one or more HTTP headers.

Based on the experience gained in the prototype, we feel that ***Sandwich*** is deficient in circumstances when two conditions on two different data elements must be met before the HTTP stream is considered to be interesting. For example, if we want to get hold of all HTML pages from all requested URLs ending with ibm.com, we will have the following. Assumed that this observing assistant is the 3rd registered assistant and we have the following entries in the *assistant.properties* file:

```
3_assistant_numSpec=2
3_assistant_spec1=com.http.HttpRequestURL,*.ibm.com
3_assistant_spec2=com.http.HttpContentType, *htm*
```

Here, for each HTTP stream, this observing assistant can potentially receive up to two events: one for the requested URL (if the URL ends with ibm.com), and the other for content type (if the content is an HTML page). The assistant remains responsible for maintaining the state information of the event and consolidating it based on the unique thread ID of the event. So, in this example, the assistant might contains two hash tables, one for keeping track of the requested URL and the other for the content type, as shown below. With this, the assistant can quickly determine the content type (from hash table B) given a service thread with a particular requested URL (of hash table A). However, the assistant has the overhead of additional processing in maintaining all these state information.



A) hash table to keep track of requested URLs

B) hash table to keep track of content type of all HTTP stream

We see that when an observing assistant is interested in more than one data element of an HTTP stream, the assistant requires some additional processing in maintaining all these state information. After constructing an observing prototype in ***Sandwich*** and conducting a detailed analysis with Webby, the lesson learned here is that the asynchronous way in notifying assistants that simply observe is a less suitable approach than the synchronous way.

Presentation Perspective: how the output of an observing assistant or Monitor MEG is propagated or displayed back to the end user.

- In ***Sandwich***, this output is presented upon request by clicking on the "View Result" button of WAMT or is pushed by the observing assistant. The presentation interface is a Java GUI that can make use of various powerful APIs such as Swing.
- In Webby, the output is presented when the user requests a pre-defined HTTP URL for the plug-in. This request will route to the appropriate Generator MEG that is responsible to consolidate persistent data stored up by the corresponding Monitor MEG. The output is then presented to the user in the form of HTML pages. These HTML pages may contain JavaScript for any required programming logic.

The two approaches mentioned above are very similar. Using Webby's approach of a pre-defined HTTP URL to get the interface of the Monitor MEG or observing assistants seems to be more consistent with the rest of the architecture because the proxy already understands HTTP. However, there are three major drawbacks to this approach:

- Use of HTML pages (optional with JavaScript) as the presentation style seems limiting; more sophisticated interface can be built when more powerful and richer GUI APIs, such as Swing, are used.

- When the amount of business logic that needs to be added increases, one is likely to end up with HTML pages with convoluted JavaScript.
- Ability to perform "push" technology is limited as HTTP supports "pull" technology.

The ideal approach seems to be to use the HTTP request approach with a full-fledge GUI application, such as applet or ActiveX component, as the front end. This removes the first two drawbacks described above. The third drawback remains but becomes feasible under the condition that the user explicitly loads the GUI application into memory the initially.

[12] The following table elaborates further on the documentation techniques used by ***Sandwich*** and Webby.

	Sandwich	Webby
Overview	• described as a framework that supports the construction of browsing • included a couple stories on the use of personal assistants, Enhanced History and Check Free Auto Post Assistant	• described as a flexible API for programming intermediaries on the web or a HTTP request and response processor • included an example of an application that transformed XML to HTML was given
Requirements	• included a set of use cases for a representative set of applications that the framework support • included a set of generalized use cases for the framework itself	• none • relied on the overview and examples
Architecture and design	• described as subsystem components • included the input/output for each type of assistants, dynamic and static view of	• included a data model for the HTTP protocol and input/output for each MEG • included a processing model on the overall flow of executions amongst

	major objects of the framework, group by their services • used the UML as the notation standard	different types of MEG
Usage	• covered concise steps on how to use the framework using the hook model	• covered with long paragraphs in describing how to use the API • referred as the programming section • relied on examples
Examples/ Prototypes	• covered with concise steps in implementing the assistant examples; these steps matched back to "changes" steps outlined in the hook(s) enacted	• covered with long paragraphs in describing how a plug-in does its job using MEGS

Webby's documentation uses the word "API" and "processor" rather than "framework" and follows the typical style in documenting APIs. Initially, this led us to a misconception that Webby was simply a library and not a framework. It was not until that we dived into the detailed documentation for how Webby operates that we realized Webby is a framework to some extent. Like ***Sandwich***, Webby does contain an inverted flow of control in invoking its plug-ins and the support to create a new plug-in as a set of MEGS.

To summarize, we feel that Webby's documentation can be further improved by

- considering the use of the word "framework" over "API" or "processor",
- including use cases for a representative set of applications that can be instantiated from Webby,
- including some static and interaction diagrams on major classes involved,
- including the hook model in documenting its usage,
- including implementation examples with steps that match those in the "changes" section of the hook(s) enacted,
- deploying UML as the notation standard

After documenting ***Sandwich*** and comparing ***Sandwich***'s documentation with Webby's, we feel that typical frameworks documentation should include, at least, the following sections:

- An **overview** section that describes what the framework is for and a brief story for its use.
- A **requirement** section that captures what the framework supports, in particular applying UML use cases to the set of representative applications of the framework. This helps to set the context in using the framework and provide an overall picture on what the framework supports.
- A section on **architecture and design** that presents
  - (a) all subsystem components of the framework based on their services,
  - (b) the object model of the framework covering the dynamics and static relationships among classes using the UML notation,
  - (c) the data model in the form of input and output for a given component as well as persistent storage involved, if any
- A section on framework **usage** (i.e. how to use the framework) using hooks.
- A section on **application examples** instantiated from the framework where their implementation details should match the steps listed in the "changes" section of the hooks involved.

It is actually very interesting and surprising to see that the end result of Webby and ***Sandwich*** are so similar because they were totally independent efforts until the recent official release of Webby Developer Kit. The first iteration of ***Sandwich*** started in the fall of 1998; Webby was evolved from the WBI Applications (note that this is an application rather than a developer kit) that was implemented in 1996 in Perl and migrated to Java in 1997. The Webby, as a developer kit, is released to the public in June 1999. ***Sandwich*** contains a default pool of three assistants; Webby's contains a default pool of five plug-in. In terms of these two aspects, ***Sandwich*** is considered to be less mature than Webby.

Here, we summarize this detailed comparison section with the following points:

- The objective of both ***Sandwich*** and Webby is the same, i.e. to provide the support in constructing personal web assistants.

- The architecture of both ***Sandwich*** and Webby relay on an intermediary personal proxy that has been granted by the user to snoop into his or her browsing activities.

- In terms of documentation, the framework-based approach taken by ***Sandwich*** has allowed it to produce better and more concise documentation than Webby. In particular, ***Sandwich*** documentation includes the use the hook model proposed by [Froe96], the application of UML use cases to a representative set of applications, and the deployment UML notation standards for static and dynamic class relationships, all of which are missing in Webby's documentation.

- In terms of design, based on the major services offered by both ***Sandwich*** and Webby, we have the following points:

  - The approach in supporting delegating assistant in ***Sandwich***, such as the use of priority queues, request and response type and default request delegate, is very similar to the approach taken by Webby's Request Editor, Monitor and Document Editor MEGS, and thus can remain intact

  - The approach in supporting observing assistant in ***Sandwich*** is slightly different from the approach taken by Webby's Monitor MEG. As mentioned before, we have learned that
    - synchronous notification to observing assistants is a better solution due to the additional processing overhead imposed by the current asynchronous notification scheme, and
    - the use of HTTP as the communication protocol with a GUI application as the front end (to get the output of observing assistants or Monitor MEG) to the proxy is a more consistent design

  - Although the administrative component of ***Sandwich*** is preliminary compared to Webby's administrative console-based component, we feel that the best approach is to deploy a web-based administrative interface to both ***Sandwich*** and Webby. Improving this component of ***Sandwich*** should be very straightforward and only requires implementation.

  - The idea of registering and de-registering assistants is the same as in both ***Sandwich*** and Webby. ***Sandwich*** can be further improved so that it does not need to be restarted for a new assistant to be activated.

6. Conclusions and Future Work

The thesis shows that having an application framework that supports personal assistants is valid and feasible. This framework encapsulates the common functionality of and is an integrated architecture for personal web assistants that have been identified and analyzed. The development of two prototyped assistants demonstrated this. New assistants are added to ***Sandwich*** by enacting one or more of its identified and documented hooks.

This thesis is also a case study on documenting application frameworks in general. By documenting ***Sandwich*** and comparing it to Webby's documentation, the two sections that we found especially useful are (a) use cases for the family of applications, and (b) framework usage documented using the hook model. The conciseness of the hook model not only can help the framework users to extend the framework but also the framework maintainers to evolve the framework. ***Sandwich*** is still in a very early stage of development and most of its hook level support are of type patterns or open-ended, and when applied, most framework features are added or replaced. This observation is consistent with [Johnson92] in that an immature framework is likely to be predominantly white-box in nature. As ***Sandwich*** continues to evolve, we foresee it shifting to a more black box framework with a pool of assistants that can optionally be hooked in. We also feel that the use of UML as the notation standard through out the documentation is also useful as it promotes consistency and understanding to a greater audience.

Sandwich can be extended and reused in the context of browsing history-dependent assistants programs. At times, we felt that this well-defined scope in targeting assistants that are based on browsing history limits some other forms of assistantship such as batch assistantship that does not require any browsing history. Nonetheless, we are grateful that this scope is enforced throughout and has helped in the completion of the project on time. Analysis on the Internet domain can go on indefinitely in this fast-changing environment as the WWW matures.

There are, of course, penalties that we pay when we deploy a proxy that does more than relays. First, there is the overhead of additional processing and therefore performance impact. Another caveat in deploying a proxy lies on the assumption made in this thesis: the end user trusts his or her personal proxy. If this assumption no longer holds in the future, further studies will need to be conducted in addressing it such as by adding some security support.

Based on the experience gained in this experiment in building ***Sandwich*** and comparing with Webby, we summarized some near future in improving ***Sandwich*** and some potential new requirements that ***Sandwich*** is anticipated to address:

- Support for synchronous notification to observing assistants

- Support for a web-based administrative interface to the framework as well as to assistants that have result panels.

- Recall that the three data elements that an observing assistant can register interest in and be notified of are HTTP headers, content, and META tags. At present, ***Sandwich*** contains implementation support for the HTTP headers only and only those headers that are needed by the two prototyped assistants have been thoroughly tested. Thus, a near term future work involves adding support for the content and META tags data elements and all the other HTTP headers using the documented **New HTTP Header Hook**.

- Support a more robust mechanism than the current rudimentary approach via direct method calls (Section 4.6.2) for assistant-to-assistant collaboration.

- At present, ***Sandwich*** does not cache or keep a copy of the HTTP content (e.g. HTML pages and GIFs). This might become a requirement later to improve performance. Determining the best approach in accomplishing this requires more in-depth evaluation, including further investigation of another active area of research, web caching. When content caching is supported, it is also expected that there is a demand for a search engine that acts upon the content cache.

- When HTTP is used with secure sockets layer (SSL), it is also frequently referred to as HTTPS. At present, ***Sandwich*** only supports HTTP and not HTTPS. More specifically, when an HTTPS communication is established, ***Sandwich*** or any HTTP-based application proxy server loses its capability to snoop on the stream. HTTPS is gaining popularity as the web evolves into a place for electronic commerce. This capability is desirable for most e-commerce web applications where confidential information such as credit cards numbers needs to be transferred over the Internet. In this situation, we foresee the need for a negotiating assistant that plays two roles. To the server, this assistant is the SSL client and to the end user's browser, this assistant is the SSL server. Thus, two SSL sessions existed, one between the browser to the assistant and the other from the assistant to the server. Further studies need to be conducted in determining the best approach for ***Sandwich*** to handle this emerging technology.

- Support for independent assistants. As we have seen in Figure 8, ***Sandwich*** was originally designed to support the group of assistants that depends on user browsing activities. Another group of assistants acts independently of user browsing activities. As this thesis progressed, we saw the need to support this group of assistants in ***Sandwich***. This requirement expands the scope of the framework and as a result will make the framework a more complete one for providing personal assistantship. The following is an example, from Professor Eleni Stroulia, where there is a need to support batch assistant in *Sandwich*.

> Here is a question for you: I want to capture a set of examples of using easySABRE http://www.easysabre.com. Basically I want to try out a set of approximately 200 combinations of cities and dates and times to travel. Could I "program" a delegate in your framework to get these traces for me? Also if the answer is yes, would the observer cache the different "screens" that come from the server locally the client side?

There are many ways to fulfill the above requirement, including the following:

a) Use ***Sandwich*** to monitor the easySABRE sites that are of interest. Using the information gathered, develop a batch program to perform the 200 combinations independent of ***Sandwich***. The Java Networking API would suffice and the program will have to cache all the response or "screens" returned from the requested sites.

b) Evolve ***Sandwich*** to support external batch assistants and allowing external batch assistants to interact with observing assistants. This approach will allow the use of an observing assistant to perform all the caching of responses from easySABRE.

7. References

[Aglets] **IBM Aglets**, http://www.trl.ibm.co.jp/aglets/index.html

[Ajanta] **Ajanta**, http://www.cs.umn.edu/Ajanta/

[BMRSS96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal. **Pattern-Oriented Software Architecture: A System of Patterns**. John Wiley & Sons, Inc. 1996.

[CacheFlow] CacheFlow, http://www.cacheflow.com/info/docs/wp/activecaching.html

[Concordia] **Concorida**, http://web.vsisinc.com/concordia/product_information/what_it_does.htm

[CSF] Garry Froehlich. Common Services Framework (CSF). http://www.cs.ualberta.ca/~garry/framework

[DAgent] **D'Agent**, http://www.cs.dartmouth.edu/~agent/

[EggGam92] T. Eggenshwiler and E. Gamma. **ET++ SwapsManager: Using Object Technology in the Financial Engineering Domain**. In Proceedings of OOPSLA 92, 1992, 166-177.

[FG96] Stan Franklin and Art Graesser. **Is it an Agent or just a Program? A Taxanomy for Autonomous Agents.** Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

http://www.msci.memphis.edu/~franklin/AgentProg.html#agent

[FHLS97] Garry Froehlich, H. James Hoover, Ling Liu, Paul G. Sorenson. **Hooking into Object-Oriented Application Frameworks.** Proceedings of the 1997 International Conference on Software Engineering, Boston, Mass., May 17-23, 1997. pp. 491-501.

[FHLS99a] Garry Froehlich, H. James Hoover, Ling Liu, Paul G. Sorenson. **Reusing Application Frameworks Through Hooks.** To appear in Object-Oriented Application Frameworks, M. Fayad, R. Johnson editors

[FHLS99b] Garry Froehlich, H. James Hoover, Ling Liu, Paul G. Sorenson. **Designing Object-Oriented Frameworks**. In the Handbook of Object-Oriented Technology. S.Zamir editor. CRC Press, New York, 1999. pp. 25-1 to 25-22.

[FHLS99c] Garry Froehlich, H. James Hoover, Wendy Liew, Paul G. Sorenson. **Application Framework Issues when Evolving Business Applications for Electronic Commerce**. Proceedings of the 32nd Hawai'i International Conference on Systems Sciences (HICSS 32), January 5-8, 1999, Maui, Hawaii. Software Technology Track. (CD-ROM), Copyright 1999 by the Institute of Electrical and Electronics Engineers, Inc. (IEEE), 10 pages.

[Fowler97] Martin Fowler. **UML Distilled: Applying the Standard Object Modeling Language**. Addison-Wesley Longman, Inc. 1997.

[Froe96] Garry Froehlich. **Hooks: an Approach to the Reuse of Object-Oriented Application Frameworks**, Ph.D. Candidacy Document, University of Alberta, 1996.

[FS97] Mohamed E. Fayad and Douglas C. Schmidt. **Object-Oriented Application Frameworks**. Communication of ACM, Vol 40, No. 10, October 1997.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley Longman, Inc. 1995.

[GM95] Gangopadhyay, D. and Mitra, S. **Understanding Frameworks by Exploration of Exemplars**. In Proceedings of 7th International Workshop on computer Aided Software Engineering (CASE-95) Toronto, Canada. 1995, pp. 90-99.

[Grand98] Mark Grand. **Patterns in Java**, Volume 1. John Wiley & Sons, Inc. 1998.

[Grand99] Mark Grand. **Patterns in Java**, Volume 2. John Wiley & Sons, Inc. 1999.

[Harold97] Elliotte Rusty Harold. **Java Networking Programming**. O'Reilly & Associates 1997.

[HHG90] Richard Helm, Ian M. Holland and Dipayan Gangopadbyay. **Contracts: Specifying Behavioral Compositions in Object-Oriented Systems**. Proceedings of OOPSLA October 1990, pp. 169-180.

[Hunter98] Jason Hunter. **Java Serlvets**. O'Reilly & Associates 1998.

[IBMEdu] IBM Java Education. **Leveraging Object-Oriented Frameworks**. Http://www.ibm.com/java/education/ooleveraing/index.html

[InfoSleuth] **InfoSleuth**, http://www.mcc.com/projects/infosleuth/publications/intranet-java.html

[JACK] **JACK**, http://www.agent-software.com.au/jack.html

[JATLite] **JATLite**, http://java.stanford.edu/

[JavaToGo] **Java-To-Go**, http://ptolemy.eecs.berkeley.edu/dgm/javatools/java-to-go/

[Johnson92] Ralph Johnson. **Documenting Frameworks Using Patterns**. Proceeding of OOPSLA '92, Vancouver, BC, Canada.

[KP88] Krasner, G. and S. Pope. **A Cookbook for Using the Model View Controller User Interface Paradigm in Smalltalk-80'**. Journal of Object-Oriented Programming, August/September 1988. pp. 26-49

[Luotonen98] Ari Luotonen. **Web Proxy Servers**. Prentice Hall 1998.

[LK94] Richard Lajoie and Rudolf K. Keller. **Design and Reuse in Object-Oriented Frameworks: Patterns, Contracts and Motifs in Concert**. Proceedings of the 62nd Congress of the Association Candienne Francaise pour l'Avancement des Sciences (ACFAS), Montreal, Canada. May 1994. Colloqium on Object-Orientation in Databases and Software Engineering.

[Liu99] Luyuan Liu. **A Tool Communicating with the Design of OO Frameworks and Hooks**. In preparation, Aug 1999.

[Nem98] Adolfo M. Nemirovsky. **Building Object-Oriented Frameworks**. Http://www7.software.ibm.com/vad.nsf/Data/Document1569. 1998.

[Odessey] **Odessey**, http://www.cs.cmu.edu/afs/cs.cmu.edu/project/coda/Web/docs-ody.html

[OROMatcher] **OROMatcher**, http://www.cs.umd.edu/users/dfs/java/

[Restina] CMU's **Restina**, http://www.cs.cmu.edu/~softagents/

[RJB99] James Rumbaugh, Ivar Jacobson and Grady Booch. **The Unified Modeling Language Reference Manual**. Addison Wesley Longman Inc. 1999.

[SG96] Mary Shaw and David Garlan. **Software Architecture**. Prentice Hall Inc. 1996.

[SOCKS] **SOCKS**, http://www.socks.nec.com/introduction.html

[Squid] **Squid**, http://squid.nlanr.net

[Srinivasan99] Savitha Srinivasan. **Design Patterns in Object-Oriented Frameworks**. IEEE Computer, February 1999. pp. 24-32

[Sun] **Sun Microsystems**, http://java.sun.com

[SY97] Wayne B. Salamonsen and Roland Yeo. **PICS-Aware Proxy System vs Proxy Server Filters**. Proceeding of INET'97, Kuala Lumpur, Malaysia.

[Tacoma] **Tacoma**, http://www.tacoma.cs.uit.no/

[W3C] W3C, http://www.w3.org/

[WebeW] WebeW, http://www.authentech-inc.com/webew/help/introduction.html

[Webby] Webby, http://www.alphaworks.ibm.com/tech/wbidk/

[WebMate] WebMate, http://www.cs.cmu.edu/~softagents/webmate/

Appendix A: HTTP

Following is a list of HTTP headers, their type and purposes sorted by the headers' names. [Luotonen98] provides a more detail description for each of these headers.

Notes:

- Header type: G for General, RQ for Request, RP for Response, E for Entity.
- The term "end client" used in the purpose description is essentially today's web browser.
- At the end of this appendix, important changes between HTTP 1.0 and HTTP 1.1 will be summarized.

Caveats:

- HTTP is still evolving, the latest specification is available from [W3C].
- Not all browsers and web servers follow the HTTP specification in their implementation.

Header Name	*Header Type*				*Purpose and Example*
	G	*RQ*	*RP*	*E*	
Accept		x			Specifies what media types are acceptable to the requesting client. E.g. *Accept: text/html, text/plain, image/gif*
Accept-Charset		x			Specify acceptable character sets. By default, all character sets are acceptable; specifying this header will narrow down the acceptable charter sets. E.g. *Accept-Charset: iso-8899-5*
Accept-Encoding		x			Specifies the acceptable encoding that the server may use. E.g. *Accept-Encoding: compress, gzip*
Accept-Language		x			Specify language preferences of the user. E.g. *Accept-language: en, fr*
Accept-Ranges			x		Indicates the web server is able to respond to "Range" request, an HTTP header that can appear in request headers. E.g. *Accept-Ranges: bytes* indicates the server support byte

					range requests
Age			x		This header's value specifies the age of the response content since the time the response was generated by the origin server.
Allow				x	The values of this header give the HTTP methods that the web server of the requested URL supports. E.g. *Allow: GET, HEAD, POST, PUT*
Authorization		x			Used to pass user's credentials to the origin server.
Cache-Control	x				Can be used to control caching in proxy servers and end clients. E.g. *Cache-control: no-cache, Cache-control: proxy-revalidate, Cache-control: must-revalidate, Cache-control: public* 1. When used in request, it indicates special request by client in guaranteeing an up-to-date response. 2. When used in response, it indicates to the origin server's instructions to proxy servers and end clients.
Connection	x				Specify communication options for the connection between the client and server. In HTTP 1.1, persistent connections are the default, i.e. connection remains open after the response has been send. This allows the client to reuse the connection. E.g. *Connection: close* will override this default, *Connection: keep-alive* (older way in doing persistent connection in HTTP 1.0)
Content-Base				x	Defines the URL that the relative URLs within the returned document are relative to. E.g. *Content-base:* http://www.hello.com/index.html
Content-Encoding				x	Indicates the encoding of the entity body of

					the response. E.g. *Content-encoding: gzip*
Content-Language				x	Identifies the language of the returned resource entity. E.g. *Content-language: en*
Content-Length				x	Specifies the length of the entity object in bytes. E.g. *Content-length: 3253*
Content-Location				x	Specifies the URL or the accessed resource and is useful when the requested URL points to a resource with multiple representation (different media type, for example).
Content-MD5				x	Contains the MD5 signature. E.g. *Content-MD5: base-64 encoded MD5 signature*
Content-Range				x	Indicates the start and end range of a byte range request together with the total number of bytes available in the entire object. E.g. *Content-range: 0-300/1000* means the first 300 bytes of a 1000 bytes object is being returned.
Content-Type				x	Specifies the media type of the object. E.g. *Content-type: text/html*
Date	x				Indicates the date and time at which the message was generated. E.g. *Date: Wed, 07 April 1999 19:40:17 GMT* 1. When used in request, it's the time the client generated the request. If proxy auto generate a request, the proxy should set this field. When used in response, it's the time the server generated the response.
Etag				x	The value of this header specifies the entity tag of the returned object. E.g. *Etag: doc-id-2441*. This is used together with *If-Match* and *If-None-Match* for object validation. It can also be used with the *Vary* header for object comparison. Format of the *E-Tag* values are

					vendor (web server) dependant.
Expires				x	Offers by HTTP 1.0 to limit caching. E.g. *Expires: -1*
From		x			Contains the requesting user's e-mail address. For privacy reason, this header is rarely sent in client request. E.g. *From: wendy@cs.ualberta.ca*
Host		x			Specifies the hostname and port number present in the URL being requested[1]. This addresses the problem of virtual multi-hosting in HTTP 1.0.
If-Modified-Since		x			Used with cache up-to-date checks to perform conditional GET.
If-Match		x			Used to perform condition request, an alternative of If-Modified-Since header.
If-None-Match		x			Inverse of the If-Match header.
If-Range		x			This header is used with byte range request.
If-Unmodified-Since		x			Used to make the request conditional, i.e. the operation (request) is carried out only if the resource has not been modified since the indicated date and time. E.g. *If-Unmodified-Since: Sun, 7 April 1999 09:30:37 GMT*
Last-Modified				x	This specifies the creation or last modification time of the object on the origin server. E.g. *Last-modified: Sun, 11 May 1997 09:30:37 GMT*
Location			x		Indicates the redirection destination, used when a 3XX redirection response status.
Max-Forwards		x			Used to limit the number of hops a request can make, used together with the TRACE method.
Pragma	x				Being phased out from HTTP 1.0 in favor of *Cache-Control* header. The only valid value

					in HTTP 1.1. is *Pragma: no-cache*. This directive is linked to the "reload" button of a browser. - When used in request, fresh pull from the server is required. - When used in response, the response should not be cached. e.g. *Pragma: no-cache*
Proxy-Authentication			x		Used with 407 proxy authentication required response code. It specifies the authentication parameters that the client should used in constructing the authentication credentials to the proxy server.
Proxy-Authorization		x			Used to pass user's authentication credentials to a proxy server. e.g. *Proxy-authorization: Basic eG1hczpjb29raWU=*
Public			x		Indicates the methods supported by the web server. E.g. *Public: GET, HEAD, POST, PUT, OPTIONS, TRACE*
Range		x			Used to make a range retrieval request.
Referer		x			Contains the URL of the document that contained the reference to the requested URL.
Retry-After			x		Used with the 503 service unavailable response status to indicate the requesting client may retry after the specified time. E.g. *Retry-after: 120* for a 2 minutes retry time.
Server			x		Identifies the server software that generated the response, e.g. *Server: Apache/1.2*.
Transfer-Encoding	x				Indicates any transformations that have been performed on the message. The only valid value in HTTP 1.1. is Transfer-Encoding: chunked
Upgrade	x				Intended for switching the protocol, or the

					protocol version, on the fly. E.g. *Upgrade: HTTP/2.0* When used in request, client indicates the protocol version it would prefer to switch to. When used in response, server indicates the protocol version it would like to switch to.
User-Agent		x			Gives the browser name and version of the end user. E.g. *User-Agent: Mozilla/4.0*
Vary			x		Used to indicate that a document is available in several languages, e.g. *Vary: Accept-language*
Via	x				Indicates the proxy chain that a request was passed through, separating by a comma. e.g. *Via: 1.1 firstProxyHost, 1.1 secondProxyHost*
Warning			x		Allows the origin server or intermediate proxy servers to attach warning messages indicating additional status information of the resource in a human-readable format.
WWW-Authenticate			x		Contains authentication parameters that client should use when preparing the authentication challenge response to an origin server.

Appendix B : META Tags

META Tags are used to embed meta information of a HTML pages. There are two main META tags: MAIN and HTTP-EQUIV. Both of these tags will proceed with the META tag and reside after the <HEAD> (or <TITLE> if present) and before the <BODY> tag. Thus, a typical HTML page with META Tags would contain the following few lines at the beginning of the page:

```
<head>
<title></title>
<meta …. >
</head>
```

An example of an HTTP-EQUIV META tag is the refresh tag used to reload or redirect a site. For example, say http://www.hello.com/page1.html contains the following META tag:

<META HTTP-EQUIV="refresh" CONTENT="2; URL=http://www.cs.ualberta.ca/page2.html">

When user loads page1.html, the page http://www.cs.ualberta.ca/page2.html will automatically be redirected in 2 seconds.

An example of a NAME META tag is to include keywords for page indexing purpose. For example, say page1 contains the following META tag:

<META NAME="keywords" CONTENT="object-oriented frameworks, personal assistants">

Subsequently, when someone does a keyword search using search engine such as www.yahoo.com, page 1 will appear on the searched list.

The attribute NAME (i.e. "keywords") refers to user-selected names, while the value for HTTP-EQUIV (i.e. refresh) means that the value has a real equivalent header in the HTTP protocol. It is important to note that META tags are useful only when the tools deployed made use of them (e.g. web browser, search engines and so on).

Appendix C: Tools and Standards Deployed

Standards Referred:

- HTTP 1.1 RFC 2068 specification

Software Tools Deployed:

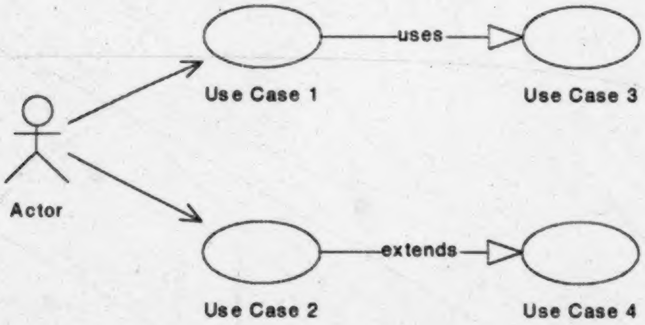
- Netscape 4.x
- Java Web Server 1.1.3
- Visual Age for Java 2 as the Integrated Development Environment (IDE) and Source Code Control
- OMG Rational Rose 98 for Java (UML)

Appendix D: UML Notation

Three major components of UML notation that are used through this thesis are summarized in this appendix. For more in depth explanation, please see [RJB99].

a) **Use Case Diagram**

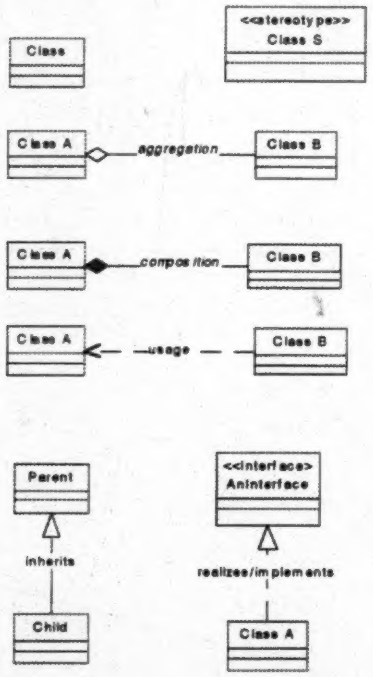
- used to capture user requirements, i.e. functionality that the system being build should provide



Use case	A collection of possible interaction between the system under discussion and its external actor(s).
Actor	A role that external entities (someone or something) in the external environment can play in relation to the system.
Uses	A uses relationship occurs when a particular behavior is similar across more than one use case and thus can be factored out and be reused by the applicable use cases.
Extends	An extend relationship is used when one use case is similar to another use case but does a little bit more.

b) Class Diagram

- used to describe static relationships among classes

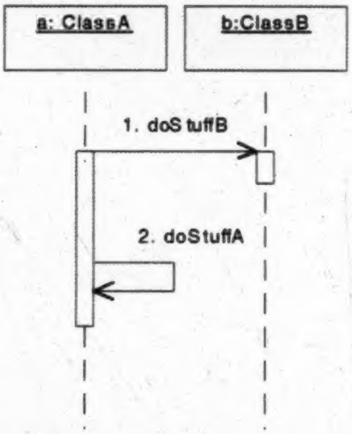

<<stereotype>>
Class S
Class
Class A
aggregation
Class B
Class A
composition
Class B
Class A
usage
Class B
Parent
inherits
Child
<<interface>
AnInterface
realizes/implements
Class A

Class	A class represents a discrete concept within the application being modeled.
Stereotype	The basic information content and form of a stereotype are the same as an existing base model but with an extended meaning.
Aggregation	An association that represents the part-whole relationship. In the above example, *Class B* is part of *Class A*.
Composition	A stronger association than aggregation in which the composite has the sole responsibility for managing its parts such as its allocation and de-allocation.
Usage	A situation where one element requires another for its correct functioning.

Inheritance	An association that represents the parent-child relationship. In the above example, the class *Child* inherits all methods, i.e. behaviors, and non-private variables of the class *Parent*.
Realize	In UML, the relationship where a class realizes a specification is referred as realization. The above notation applies in Java where most specification is included in an (Java) interface and a class realizes it by implementing this interface.

c) Sequence Diagram

- use to describe the interaction (dynamic view) of collaborated objects in the system



- "a: ClassA" means that "a" is an instance of "ClassA"
- "step 1 for doStuffB" means object "a" invoke the *doStuffB()* method of object "b"
- "step 2 for doStuffA" means object "a" invoked one of its own method *doStuffA()*.
- vertical axis can be labeled as time; horizontal axis can be labeled as objects

Appendix E: Sandwich Properties Files

a) File: *sandwich.properties*

```
#
# All the configurable factory parameters that Sandwich uses
#
admin_application_classname=com.admin.WAMT
logger_classname=com.logging.LoggerStdout
regular_expression_classname=com.common.FastMatcherAdapter

#
# The file that contains which assistants are hooked in the fw
#
assistant_filename=assistants.properties

#
# The port at which the proxy component will listen to
#
port_number=8082
```

c) File: *assistant.properties*

```
numAssistants=3

1_assistant_classname=com.assistant.pool.enhancedHistoryAssistant.EnhancedHistoryAssistant
1_assistant_name=Enhanced History Assistant
1_assistant_activeObserving=1
1_assistant_description=This assistant will monitor all sites you have visited and produced a statistic report out from it.
1_assistant_metaDataFileName=EnhHistory.prop
1_assistant_numSpec=2
1_assistant_spec1=com.http.HttpRequestURL,.
1_assistant_spec2=com.http.HttpResponseCode,200|304|404

2_assistant_classname=com.assistant.pool.autoPost.AutoPostAssistant
2_assistant_name=CheckFree AutoPost Assistant
2_assistant_activeObserving=1
2_assistant_activeReqDelegating=1
2_assistant_ReqDelegatingPriority=1
2_assistant_description=This auto post assistant will prompt user whether an auto post on forms that have been previously filled out.
2_assistant_metaDataFileName=AutoPost.prop
```

```
2_assistant_numSpec=3
2_assistant_spec1=com.http.HttpMethod,POST
2_assistant_spec2=com.http.HttpRequestURL,http://qs.secapl.com/cgi-bin/qs|http://qs-alt.secapl.com/cgi-bin/qs
2_assistant_spec3=com.http.HttpRequestBody,.

3_assistant_classname=com.assistant.DirectHttpAssistant
3_assistant_name=Direct Http Assistant
3_assistant_activeReqDelegating=1
3_assistant_ReqDelegatingPriority=2
3_assistant_description=Default delegate of Sandwich.
3_assistant_metaDataFileName=DirectHttpMD.txt
3_assistant_numSpec=0
```